Interactive 3-D Visualization of Feature-traces

Master Thesis
Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Christoph Wysseier

November 2005

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz
Orla Greevy
Prof. Dr. Michele Lanza

Institut für Informatik und angewandte Mathematik

The address of the author:

Christoph Wysseier
Länggassstr. 74
CH-3012 Bern
chris@wysseier.net

Abstract

The maintenance or reengineering of an object-oriented system includes its reverse engineering. In other words its internal structure and behavior needs to be understood. Many researchers have proposed different techniques to support the reverse engineering effort. The two predominant approaches to reverse engineering are:

- static analysis of source code
- dynamic analysis of behavior of a system at execution time.

Both techniques have strengths and short comings. The static analysis of an object-oriented software system face difficulties such as polymorphism and it may be difficult to correlate parts with system functionality. Dynamic analysis approaches generally do not achieve full coverage of a software system. Moreover, due to the volume of data captured during dynamic analysis of a system, it is difficult to infer high-level views.

In this thesis we address this issue of software maintenance and reengineering and propose a novel visualization technique which combines static analysis of source code with dynamic information extracted by exercising features of a system. We refer to these as *dynamic feature-traces*. This technique supports the software engineer in understanding the behavior of software systems by visualizing it in terms of its internal structure. Using our visualizations we focus on modeling features of a system, how to detect and locate them in the source code, hotspots of behavior and feature interaction as a means to understand how different features behave in a software system.

Acknowledgements

I would like to thank everybody from the university who supported me in one way or another to finish my studies by writing this thesis. First of all, thank you Prof. Dr. Oscar Nierstrasz that you welcomed me in your research group. It was a pleasure working with all the people you are surrounded with.
Two of them I would like to thank especially. The main vision which lead to this thesis came from my friend and supervisor Michele Lanza. Thank you very much, Michele, for leading me in this direction and also for the beautiful flat I could take over. I wish you all the best for your work at the Università della Svizzera italiana. My supervisor in Berne was Orla Greevy who became very important for my master thesis. Orla, the result of you work and your personal help for my thesis was an important and at the beginning not expected support. Thank you very much for everything and good luck for your own studies.
Of course I met during my studies a lot of other students. To list all of them would lead to an endless list. So I would like just to thank you all for being my friends and I wish you all good luck for your future.

During the last five years I enjoyed the support of my family and a small but important circle of friends. Thanks to my parents and my little sister for helping me through these sometimes difficult times. A special greeting goes to my circle of friends in the eastern part of Switzerland. Ueli Niederer, Reto Fischer *et.al.*, the weekends there were always very recreative. Thank you! Of course I would also thank all the friends here in Berne: Markus Beyeler, Daniela Luginbühl, Maurus Bärlocher, Markus Kobel, Adrian Lienhard, Dominik Wyss and Anne Simon to name just a few of them. A special regard goes to Anna Schmid. Without her I would not be at this point personally and professionally as I am now. And last but not least all the guys from the team handball club with which playing a team sport makes a lot of fun.

A lot of my thoughts go to my little godchild Nayana Solomita and her parents Renato and Jasmin. Thank you to entrust me this honor and for being my friends. Your independent view on my life is refreshing and has helped me to almost always only look forward.

A big part of my life became netstyle.ch, the company I founded together with Adrian Lienhard. Together with him and a lot of cool and professional co-workers we realized something which I have never thought of at the beginning although it was not always easy. This company will probably influence my future for a long time. Thank you everybody, namely my co-workers, customers, and other supporters, for the possibility to work on this dream.

Christoph Wysseier
November 2005

Contents

Chapter 1

Introduction

Sommerville [**?**,] and Davis [**?**,] estimate that the cost of software maintenance accounts for 50% to 75% of the overall costs of a software system. It would thus seem advisable to rewrite software as soon as it does not fulfill the requirements anymore. However, it may be that the software is too valuable to be replaced or to be rewritten. By adapting such *legacy systems* to new requirements the lifetime of a system can be increased which increases the return of investement for their owner. Reengineering is therefore an integral part of the lifetime of a software system if it evolves to meet new and changing requirements.

Reengineering a software system is a difficult task and complicated by many factors. Original developers may not be available anymore, the documentation is outdated or not available at all and the quality of source code may have degraded over time as a result of continuous software maintenance. Therefore reverse engineering is a key task during reengineering software systems.

Reverse engineering of software systems is part of the reengineering life cycle. It is defined by Chikofsky and Cross as "the process of analyzing a subject system to identify the system's components and their relationship, and to create representations of the system in another form at a higher level of abstraction" [**?**,]. It is the prerequisite for the maintenance, reengineering and evolution of software systems. Since modifications of one part of a system may impact other parts of the system it is essential to have a mental model of the software before the system can be modified or reengineered.

We have focused our attention on the reverse engineering of *object-oriented* legacy systems, mainly because most current software systems are written in languages implementing this paradigm. Reverse engineering object-oriented software systems comes with additional challenges [**?**,] compared to non-object-oriented systems, such as polymorphism, late-binding, incremental class definitions, etc.

We identify the two predominant approaches that address the task of reverse engineering object-oriented software systems:

- **Static source code analysis**
  In this approach a model of the source code is abstracted to support the understanding of the internal structure and design of a software system in terms of source code artifacts.
- **Dynamic behavior analysis**
  In this approach information about method invocations, state, etc. of the running software system is collected and interpreted to understand the behavior during execution.

Many reverse engineering approaches to software analysis focus on the source code artifacts of a system, such as classes and methods. A static perspective considers only the structure and implementation details of a system. Using static analysis alone we are unable to easily determine the roles of software entities play in the features of a system and how these features interact. We define a feature as a unit of observable behavior of a system [**?**,]. Without relationships between features and the software entities that implement them it is difficult to determine if a maintenance change cause undesirable side effects in

other parts of the software system. In this thesis our main focus is on the dynamic behavior analysis of features to support the understanding of the run-time behavior of features.

1.1 Dynamic Behavior Analysis and Software Reverse Engineering

In the context of software reverse engineering dynamic behavior analysis is used to understand the run-time behavior of a software system. Demeyer *et.al.* describe a reverse engineering pattern that recommends stepping through the execution of the software system to gain an understanding of the dynamic behavior of a software system [**?**,]. This pattern underlines the motivation and the problem of dynamic analysis. The motivation is to automate the analysis of run-time behavior because stepping through the execution manually is very time-consuming. Another problem is the collection of the information especially because of the vast amount and complexity of the data gathered. To reduce the data researchers often focus on features. This allows to only focus on the parts of the software system that are affected by the execution of a feature. Moreover, different techniques were developed to filter the data using common subtree approaches, concept analysis, clustering techniques, etc.

To obtain the execution trace we instrument the code using method wrappers [**?**,]. We collect each method invocation that occurs during the execution of a feature. We refer to this execution traces as *feature-trace* which represent the run-time behavior of features.

One focus of feature-centric analysis approaches is to correlate features and source code entities. By understanding which parts of the code are providing functionality to the features, we support the maintenance task as we identify which parts of the system may be affected by a maintenance change. Greevy *et.al.* [**?**, **?**,] outline a feature approach to reverse engineering whereby they exercise a set of features and establish how the classes or methods relate to these features. One of their visualization techniques support the identification of parts of the system which are active in one or more feature or inactive. By obtaining dynamic views of a software system at run-time and characterizing the roles of the software entities this information also allows to speculate about the design of the software. Identifying the features that are using the same parts of the system may indicate that these parts of the system have been implemented in a generic manner.

Apart from the identification of parts of the system which are used by one ore more features it is also important to identify which parts are more or less active during the execution. This information may be used to optimize the behavior of features and reveals information about the architecture of the software system.

1.2 3-D Visualization of Dynamic Behavior

We introduce a novel visualization technique based on 3-D visualization. Mainly, our goal is to support the reverse engineer to understand run-time behavior of features by visualizing each message invocation that occurs within a feature-trace. Therefore the individual method invocations collected during the execution of features are interpreted and visualized. As the basis of our visualization we use 2-D polymetric views and extend this approach by modeling object instantiation as first-level entities and visualizing these with the third dimension of the views. The visualization supports the understanding of a feature execution. Futhermore, this approach provides an easy way to detect how features correlate with the software system and relationships between features to reveal patterns of execution. It also provides a technique to detect parts of the system which are stressed most.

We realize our approach as an enhancement of the existing *Moose* reengineering environment [**?**,]. We integrate existing tools such as *TraceScraper* [?], *CodeCrawler* [**?**,], etc. to collect and visualize the information. Like this we take advantage of existing capabilities these tools are providing and are contributing new approaches. In the following sections we describe the goals of this work and some of the contributions of our approach.

1.3 Goals of this Work

Applying our visualizations of the dynamic behavior of features in terms of object instantiations and method invocations we would like to answer the following questions:

- Does our 3-D visualization of feature execution support program comprehension of the dynamic behavior?
- Which parts of a software system are affected by one or more features?
- How do the features interact with each other? Which parts of the system are used by all the features?
- Can we identify patterns of activity that are shared by features?
- Are there any parts of the system that are stressed? By stressed we mean areas of high activity in the execution of a feature, in other words classes and objects that are sending and receiving a lot of messages.

1.4 Contributions

In this thesis we present a novel 3-D visualization to understand dynamic behavior of features and their correlation to the static structure of the software system. The contributions in detail are:

- We extend the technique of method wrappers to get an unique identity of object instances contributing to a feature. This allows to explicitly determine which software entities such as classes and objects are communicating with others and at which point of the feature-trace they were created. A detailed description of this approach is discussed in Chapter **??**.
- We provide a new visualization technique to analyze feature-traces using the combination of dynamic and static software analysis. Mainly, we introduce the *dynamic feature-trace view* which visualizes the behavior of a feature at a specific point of a feature-trace. We discuss this technique in Chapter **??**.

- We extend the visualization of feature characterization of classes by providing a *static feature interaction view*. This view maps the feature class characterization measurement [?] to a color of the 3-D polymetric view [**?**,] of the *system complexity view*. Our view indicates which classes are characterized with respect to features. The principles of this view are shown in Chapter **??**.

- We describe a technique to detect *feature hotspots*, *i.e.* software entities that are more active than others during the execution of a feature. To achieve this we use a static representation of the *dynamic feature-trace view*. We introduce the detection of *feature hotspots* in Chapter **??**.

- Using our case studies in Chapter **??** we show how our approach supports the understanding of the run-time behavior as well as its collaboration with the static structure. Therefore we introduce a methodology to lead a software reverse engineer using our tools step-by-step.

- We build a tool *TraceCrawler* which allows to step through the behavior of a feature using the *dynamic feature-trace view* and which controls the visualization. We describe this tool in Appendix **??**.

- We build an interactive 3-D visualization engine called *CCJun* which provides the interface to create 3-D views and navigation functionality to move through the 3-D space and zoom. Furthermore it offers an interface to the meta-model of the *Moose* reengineering environment to provide more detailed information about the entities displayed on screen. We provide a detailed discussion of all tools in Appendix **??**.

1.5 Thesis Outline

- In Chapter **??** we introduce the analysis of feature-traces and show some state-of-the-art analysis techniques. We identify and discuss the scope and limitation of this research area. We then outline our 3-D visualization approach and introduce our feature views.

- We present the novel visualization technique in Chapter **??**. We explain the principles of this visualization and its use in the area of reverse engineering software systems.

- In Chapter **??** we present the results of applying our visualization to two case studies as a proof of concept of our approach. We apply our technique to the systems *SmallWiki* and *Moose* and describe our methodology of analyzing the *feature-traces* based on these examples.

- In Chapter **??** we discuss the results we obtained from applying our approach, list some limitations of our approach and identify future work.

- In Appendix **??** we describe in detail the tools which provide the foundation of our work.

- In Appendix **??** we provide a developers guide to describe how to use *TraceCrawler* in the context of *Moose* and *TraceScraper*.

Chapter 2

Analysis of Dynamic Feature-traces

As outlined in the previous chapter there are predominantly two areas of analysis that focus on software comprehension, namely the static source code analysis and the dynamic behavior analysis. Many reverse engineering approaches to software analysis focus on static source code entities of a system, such as classes and methods. They use a wide range of tools such as visualizers, query engines and techniques such as visualization, clustering, concept analysis, etc. which generate a high-level view of the source code with different focus on the target software system. The *Moose* reengineering environment [?] provides a wide range of such software reverse engineering tools and techniques. It provides a lot of different software metrics, querying language and navigation support for the software engineer. Furthermore there exist various tools that use the FAMIX meta model of *Moose* [**?**, **?**,] for other purposes such as software visualization [?], concept analysis [**?**,], etc.. But because we are unable to easily determine the roles of software entities play in the features of a system and how these features interact using static source code analysis only we focus in this thesis on dynamic behavior analysis of features.

In the case of dynamic analysis large traces complicate the task of generating high-level views. Many researchers have focused on reducing and compressing traces. Using visualization, filtering, compressing or other techniques to reduce the information the reverse engineer should be able to understand the dynamic behavior of a software system. A further technique to reduce the amount of information is to focus on features, *i.e.* a unit of observable behavior of a system. Because a full coverage of the software system is impossible Greevy and Ducasse show that a feature-driven approach is sufficient to detect which parts of the code are participating in a set of features [?]. Exercising features on a software system generates an execution trace which we refer to *feature-trace*. Because of its vast amount and complexity the collection and interpretation of *feature-traces* is a difficult task.

In this chapter we introduce the collection and interpretation of feature-traces as well as standard techniques. Furthermore we describe how software visualization can be used to generate high-level views of software system and finally present our approach which combines these methods to a novel visualization technique.

2.1 Feature-trace Collection

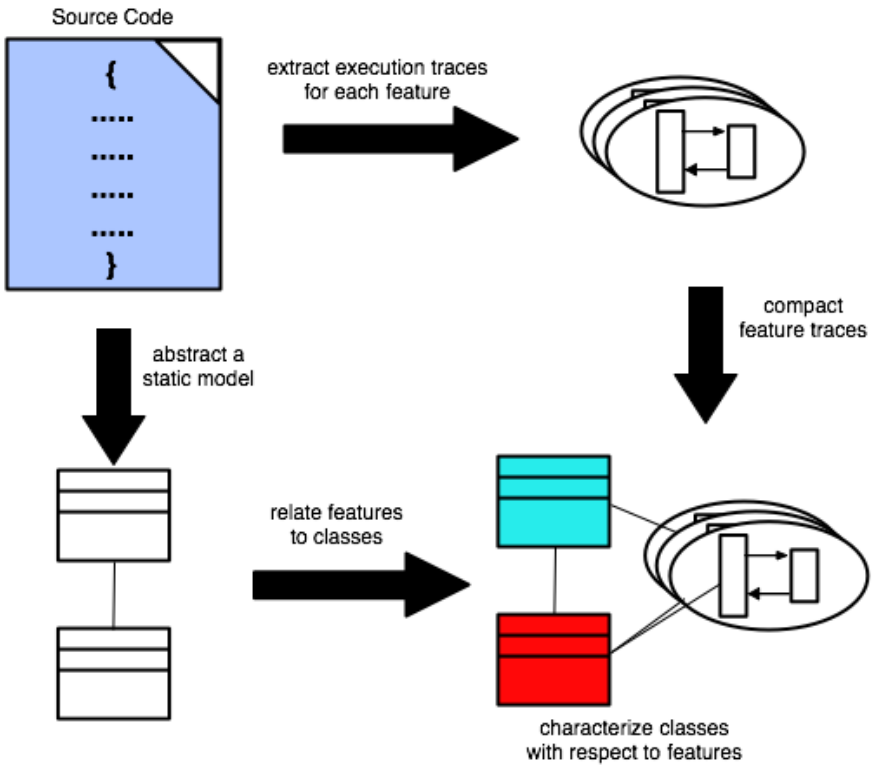

Figure 2.1: An illustration showing the collection and correlation of *feature-traces* with static entities using dynamic and static analysis

The basis of our dynamic behavior analysis is the information that is collected during the execution of a feature. We model features as test cases which are then executed in a instrumented environment. The code instrumentation is done using method wrappers [?] which are installed in the target software system. During exercising a feature every method call is extracted which leads to a collection of method invocations, the so called *feature-trace*. The elements of the *feature-trace* represent a specific method call with additional information such as the sender and receiver object/class. The upper right part of Figure **??** shows the *feature-trace* extraction and its compaction to a feature set.

Code instrumentation is a difficult task in different programming languages. One needs to install method wrappers which capture every method invocation. This requires an architecture where each method invocation is executed within one method in the systems core. If this premise is fulfilled, the method wrappers slow down the execution of the feature which may lead to a problem in case of large features. Besides, this approach yields to a vast amount of information that is gathered from the software system. An abstraction and high-level interpretation is therefore crucial to extract useful information about the run-time behavior of features.

2.2 Feature-trace Interpretation

To support comprehension of the run-time behavior we propose to analyze *feature-traces*. There are a variety of approaches adopted to manipulate the large volume of trace information and abstract high-level views.

- **Software Visualization:** As listed below several approaches are using visualization to interpret the *feature-trace*. A simple visualization is to display the *feature-trace* as a tree of method invocations. It is obvious that this technique is difficult to handle with large *feature-traces* because an overview is difficult to obtain.
  The UML standard [**?**,] provides four kinds of behavioral models: sequence diagrams, collaboration diagrams, state diagrams and activity diagrams. Using a sort of sequence diagram one may obtain a view of the *feature-traces* as provided by ISVis [**?**, **?**,]. Program Explorer [**?**,] offers also simple sequence diagram and collaboration diagram like layouts intended for displaying only small parts of a *feature-trace*.
  De Pauw *et.al.* [**?**,] represent execution traces using a variation of an interaction diagrams [**?**,]. They handle the complexity by condensing the information using a zoom functionality to provide an overview of the trace. Walker *et.al.* [**?**,] display the interaction between objects using program animation techniques. Their tool focuses on displaying the number of objects involved as the execution progresses. A more coarse-grained view proposed by Antoniol *et.al.* [**?**,] visualizes the evolution of features in combination with the static software entities.

- **Software Metrics:** With this technique the collected information is compacted by applying measurements, for instance the frequency of calls or the number of objects. The results are usually represented visually [**?**, **?**,].

- **Filtering and Clustering:** In this strategy the amount of information to be displayed or analyzed is reduced using filtering and clustering techniques. ISVis [?, ?] provides filtering and clustering technique to reduce the information before visualizing the execution traces. Eisenbarth *et.al.* [**?**,] apply formal concept analysis to reduce the amount of information and are generating high-level views of its result.

Nevertheless an accurate interpretation of the execution traces itself is difficult because of the huge amount of data. Displaying a large trace as a sequence diagram produces a complex diagram where an overview is impossible. This is also caused by the large amount of objects of different classes that are affected by the execution trace.

Combination with Static Source Code Analysis
To detect undesirable side effects caused by a maintenance change on a feature it is important to determine what role static software entities play in a feature. Several works have shown that a feature-driven approach allows to limit the amount of information to the parts of code that are important [?]. It is essential to establish a relationship between features and the source code entities which has been proposed by several researchers with different techniques such as concept analysis, visualizations, etc. [**?**, **?**, ?,].

Greevy *et.al.* describe an approach based on software metrics and proposes characterizations of features and classes by how they participate in features [?]. Figure **??** shows their approach as an illustration. By analyzing the *feature-trace* for each feature that is examined one may determine which class is active in one or more feature or which class are inactive with respect to the features traced. On the other hand a feature can be characterized by a so called *feature-fingerprint* which is a set of classes which participate in the feature. This mapping between features and static software entities supports program comprehension and facilitates maintenance changes. Moreover, in a previous work we introduced our *static feature interaction view* which reveals the parts of the software system that participate within one or more feature [**?**,].

Richner [**?**,] has also conducted research on the combination of static and dynamic information with the goal of recovering behavioral models of a software system. Therefore Richner uses a query-based approach using perspectives which is a model of the kind of dynamic information a software engineer is interested in. The perspectives are defined using queries on the source model.

2.3 Software Visualization

Software visualization is a specialization of information visualization where all is about reduction of complexity. Software visualization is defined as ”the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction and computer graphics technology to facility both the human understanding and effective use of computer software” [**?**,]. The goal is to provide a view on the software system on a higher level of abstraction which supports the reverse engineer in understanding the software system.

In the context of static source code analysis information that can be extracted from the static structure of the software system is visualized. Many tools make use of static information to visualize software, like Rigi [**?**,], Hy+ [**?**,], SeeSoft [**?**,], ShrimpViews [**?**,], GSee [**?**,], and the FIELD environment [**?**,], to name but a few prominent examples.
Lanza *et.al.* propose the polymetric view [?] which provides a visual overview about the design and a validation possibility about the design speculations made during the first contact with the system. Based on the visualization one may identify exceptional entities, detect design patterns implemented or design problems.

In the context of dynamic behavior analysis visualization is used to provide a high-level view of feature-traces. Many researchers proposed such views to reduce the amount of information and complexity of a feature-trace as we showed in the previous section.

2.3.1 Software Metrics

Software metrics measure certain properties of a software system by mapping them to numbers. They are widely used to assess the quality and complexity of software [**?**,] and in recent years metrics have been defined and applied to object-oriented software as well [**?**, **?**,]. This simple approach scales up for large software systems and is language independent. In case of simple metrics they profit from their reliable definition. However, simple measurements are hardly enough to sufficiently and reliably assess software quality [**?**,].

Most of the metric tools visualize information using diagrams for statistical analysis, like histograms and Kiviat diagrams. Datrix [**?**,], TAC++ [**?**, **?**,] and Crocodile [**?**,] are tools that exhibit such visualization features.

2.3.2 Polymetric View

The polymetric view [?] is a visualization of static code enriched with up to five software metrics. Basically the nodes are representing software entities while as the edges are representing relationships between them. This method of visualization is then enriched with the metrics by adapting the node size, color and position.

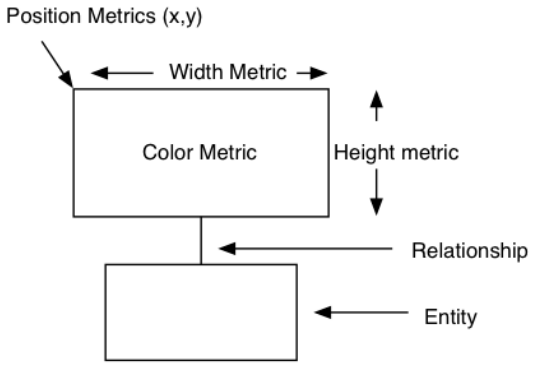

Figure 2.2: The principle of polymetric views

This approach combines software visualization with software metrics and therefore eliminates the need of interpretation huge metric tables. After learning the visual language of polymetric views one gets a fast overview about the software system, its entities and relationships.

2.4 Our Approach

Our approach is an extension of existing approaches such as feature-driven dynamic analysis and software visualization. We are analyzing the *feature-traces* collected from a software system and are visualizing and animating these to support the understanding of run-time behavior of a software system as shown in Figure **??**. Below we list some key points of our approach:

- **Feature-centric approach:** We focus on features of a system and visualize the *feature-trace* in the context of a static entities of a system such as classes and methods. We model a feature as a tree of method invocations. Although our *feature-traces* do not achieve a complete coverage of the system under analysis, our feature perspective helps the software developer to focus on specific parts of the code that would be affected by mainenance change to these features.

- **Filtering:** To reduce the amount of information visualized we propose two filtering techniques. On the one hand it is crucial to focus on selected parts of the *feature-trace*. We introduce a simple filter which enables us to focus on selected parts of interest of a trace. Furthermore we filter the message invocations between objects and their meta-classes that are concerned with the initialization of newly created instances.

- **3-D Visualization and Animation:** In the research area of software reverse engineering 3-D vs. 2-D visualization there is a controversial debate in progress [**?**,]. For our approach we choose the 2-D polymetric view and exploit the third dimension to visualize object instantiation. We extend the polymetric views by two further properties. The depth of an object which leads to quaders instead of rectangles representing software entities and the position on the z-axis which we use to build towers of quaders. In case of our visual metaphor a 3-D visualization is useful because we are able to add additional information without the need to limit the approach of polymetric views. To enhance the interpretation of *feature-trace* we propose animated visualizations which allows the

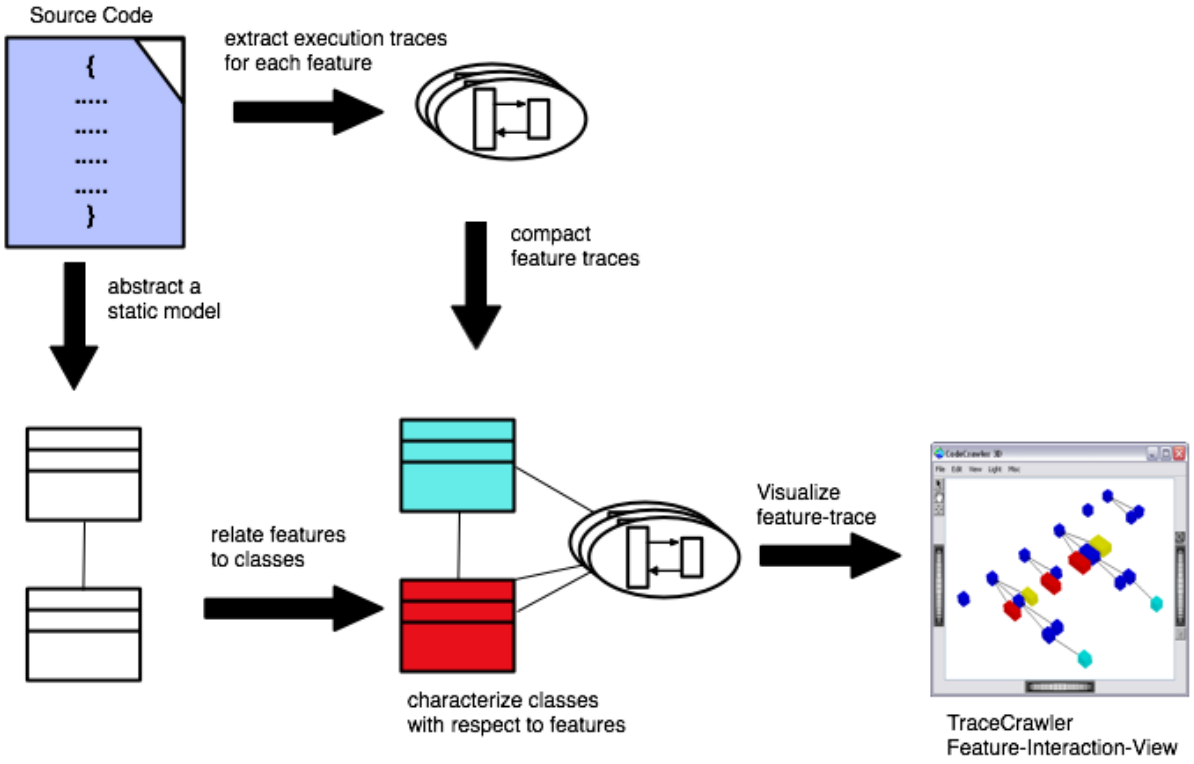

Figure 2.3: An illustration of our approach extending feature-driven analysis with software visualization

software engineer to analyze the state of a *feature-trace* at a specific point of execution. We propose to step through the *feature-trace* selectively to understand the run-time behavior of a feature. Using this technique the software engineer can also detect the formation of *feature hotspots*, i.e. object instances or classes which are stressed during the execution. Moreover, we propose several static views which support the comprehension of how a feature or parts of it affects specific parts of the software system or other features.

- **Software Metrics:** To support the understanding of the interaction of features we use the FC measurement as a color metric for our polymetric views. This metric characterizes a class in terms of how it collaborates within one or more features [?].

- **The Combination of Static Source Code and Dynamic Behavior Analysis:** Our novel approach combines the static source code and dynamic behavior analysis within one single visualization. The main advantage of this technique is that several research questions can be answered within one view. We map features to software entities such as classes but also to object instances which are instantiated during the execution of a feature. Moreover, the proposed visualization provides information about the static structure of the software system as well as information about the method calls out of a *feature-trace*.

In the following chapter we will introduce our approach in detail. We show how we collect and interpret *feature-traces* and generate our static and dynamic views to analyze them. Moreover, we introduce a methodology of analyzing our views and present in a further chapter the results of applying our approach to two case studies.

Chapter 3

Interactive 3-D Visualization of Feature-traces

In this chapter we introduce our approach of analyzing *feature-traces* which is intended to support the software reverse engineer to understand the run-time behavior of features. Our approach is a novel visualization technique that combines the dynamic information collected exercising features with the static structure of the software system. This way we realize a visualization which provides an abstracted and compacted view on the *feature-traces* and allows to reveal how features correlate with the source code.

As a start we describe the application of the extended method wrappers. Using code instrumentation our tool *TraceScraper* extract the *feature-traces* that are loaded into the visualization engine. Furthermore, we explain how these information are stored within our *Moose* reengineering environment.

We describe in detail the visualization of dynamic information which we refer to as the *dynamic feature-trace view*. Using this technique the software developer can selectively step through execution traces and drive a visualization engine which displays a 3-D representation of the events of a feature in terms of object instantiations and message invocations between objects. Therefore the developer can see how the system behaves during the execution of the traces. Moreover, the visualization is interactive and navigable, *i.e.*, the user can examine in detail interesting objects and also change his point of view in the 3D space to get a closer look at specific parts of the system being traced. The tool which implements these dynamic feature-trace views is called *TraceCrawler*, an extension of the *CodeCrawler* tool [**?**, **?**,] and *CCJun* [**?**,].

TraceCrawler also provides static views which we use to detect *feature hotspots*. The term *hotspot* is used in many different contexts. In a geological context for example, a *hotspot* is used to refer to areas of volcanic activity. According to Wikipedia[1] *Hot spots* are defined as areas of high activity that are surrounded by areas of lower activity. In the context of feature analysis, we use the term *feature hot spot* to refer to areas of high activity in a system during the execution of a feature which we reveal using our visualizations. We therefore analyze the feature-traces to detect *feature hotspots*. We consider objects that appear as central points of communication to be *feature hotspots*. In other words they send and receive a higher than average number of messages than other instances.

[1] See http://www.wikipedia.org

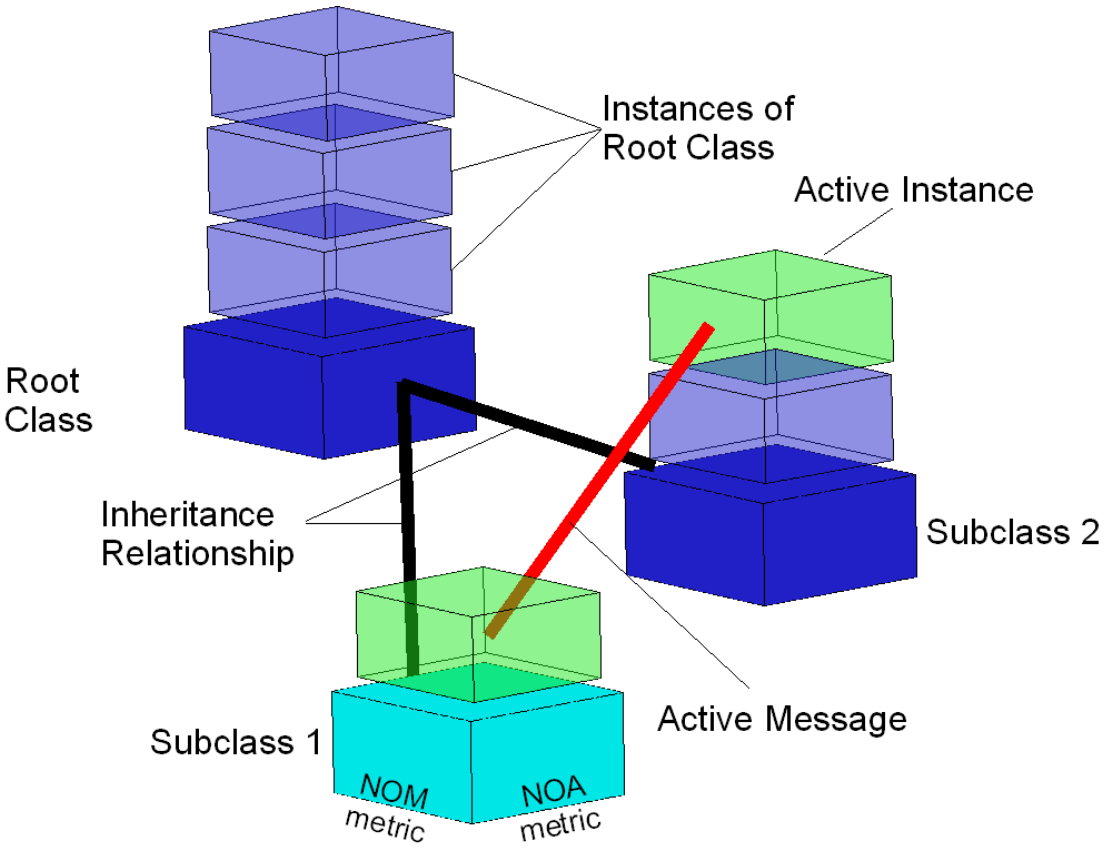

Figure 3.1: The principles of the *dynamic feature-trace view*

3.1 Understanding Dynamic Behavior of Features

3.1.1 Feature-trace Collection

As a basis of our visualization technique we need detailed information about the run-time behavior of the software system. Therefore our tool *TraceScraper* installs method wrappers which record each message invocation while exercising a specific feature. The method wrappers record the following information for each event:

- **Sender:** The name and a unique id of the sending object or class which executes the method
- **Receiver:** The name and a unique id of the object or class where the call is executed
- **Return value:** A unique id of the object or class that is passed as return value from the method currently executed

Each of these events are stored within a collection and classified into two groups. We distinguish between constructors which instantiate new instances and other ordinary message invocations. When the new instances are created the return value of the constructor is the unique id which identifies the new instance. This is necessary to map later on the sender and receiver of message invocations to the appropriate instances.

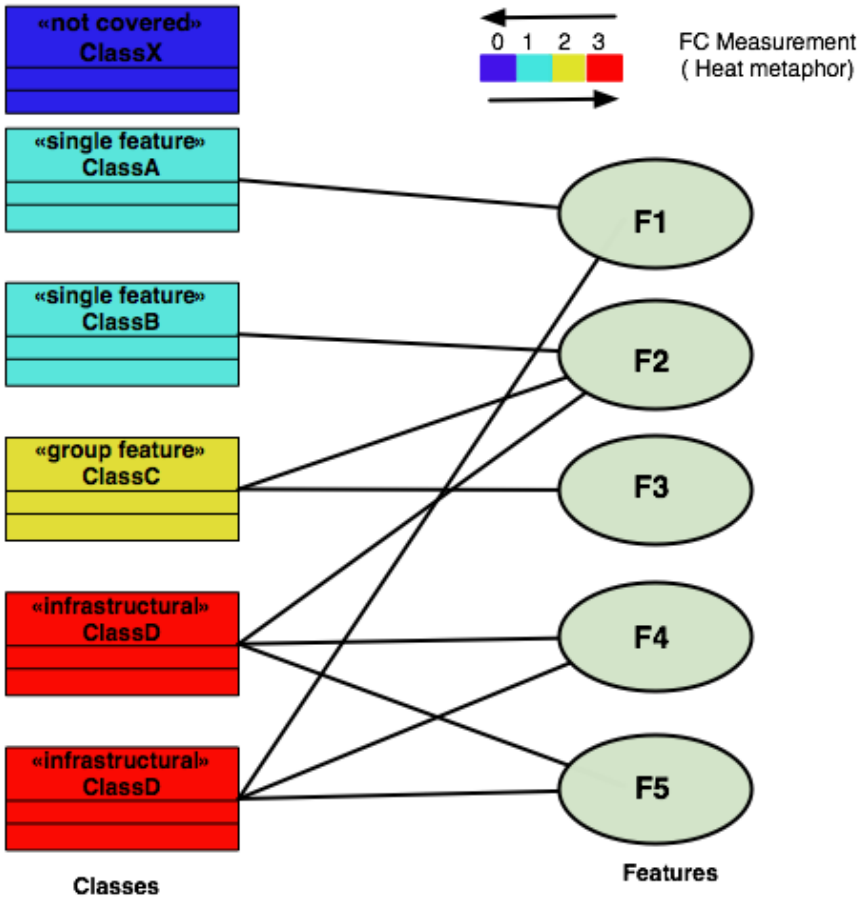

Figure 3.2: Class characterization using the FC measurement

3.1.2 Navigable Visualization of Feature-trace

The *dynamic feature-trace view* is a representation of the behavior of a system during the execution of a feature in terms of classes, object-instantiations and message sends. In Figure **??** we see a schematic display of such a view: It is a 3-D visualization which displays the static structure of the system on a plane floating above the ground. The boxes on the ground are the classes connected by black edges representing inheritance. This way of representing the static structure of a software system was introduced with polymetric views [?]. We use an adapted *system complexity view* for our visualization which is provided by our tool *CodeCrawler* [?] in 2-D and extended to 3-D by *CCJun* [?]. The position of the boxes are computed using the vertical tree layout. The view uses the NOA (number of attributes) as a measurement for the width and NOM (number of methods) as a measurement for the depth of the boxes as shown in Figure **??**.

The color of these boxes is computed by the FC measurement. This measurement computes the characterization of a class by counting how many features reference it and assigning it a value to represent the characterization below. Figure **??** shows an illustration of how classes were characterized. The red class participates within all features $F1$ - $F4$ and is therefore infrastructural. On the other hand the two classes on the top do not participate within a feature at all and are therefore *not covered* by the features. We color the nodes according to the heat metaphor presented in Figure **??**. Classes that participate in no feature are colored in blue and classes that participate in over the half of the features are shown in red.

- *Not Covered (NC)* is a class that does not participate to any of the features-traces of our current feature model.

$$(NOFC = 0) \rightarrow FC = 0 \rightarrow blue$$

- *Single-Feature (SF)* is a class that participates in only one feature-trace.

$$(NOFC = 1) \rightarrow FC = 1 \rightarrow cyan$$

- *Group-Feature (GF)* is a class that participates in less than half of the features of a feature model. In other words, group-feature classes/methods provide functionality to a group of features, but not to all features.

$$(NOFC > 1) \wedge (NOFC < NOF/2) \rightarrow FC = 2 \rightarrow yellow$$

- *Infrastructural (I)* is a class that participates in more than half of the features of a feature model.

$$(NOFC >= NOF/2) \rightarrow FC = 3 \rightarrow red$$

Figure **??** shows the *Root Class* and *Subclass 2* are infrastructural classes, i.e., are used by more than the half of the features. *Subclass 1* on the other hand participates only in one single feature.

Our color view also make use of colors to represent instances. When the *feature-trace* is interpreted step-by-step each instantiation of a class (the creation of an object) generates a blue box (like a floor in a building) above the ground level which is the appropriate class. The more blue boxes that are above a class, the more instances of this class have been created during the execution of the feature. We refer to this phenomenon as a *tower of instances*. The currently active objects are displayed in green. Each time an object sends a message to another object, a red message edge is drawn between the two object boxes.

During the reverse engineering process we get a high-level view of the system and then focus on a part of the system of interest. Zooming into the visualization and using the context-sensitive menus the user is able to identify the class and to obtain more fine-grained information about the software entities of interest. The navigation allows to zoom (or fly) to the wished part and using rotation one can change the viewpoint to get a better interpretation in case of a lot of objects on the screen.

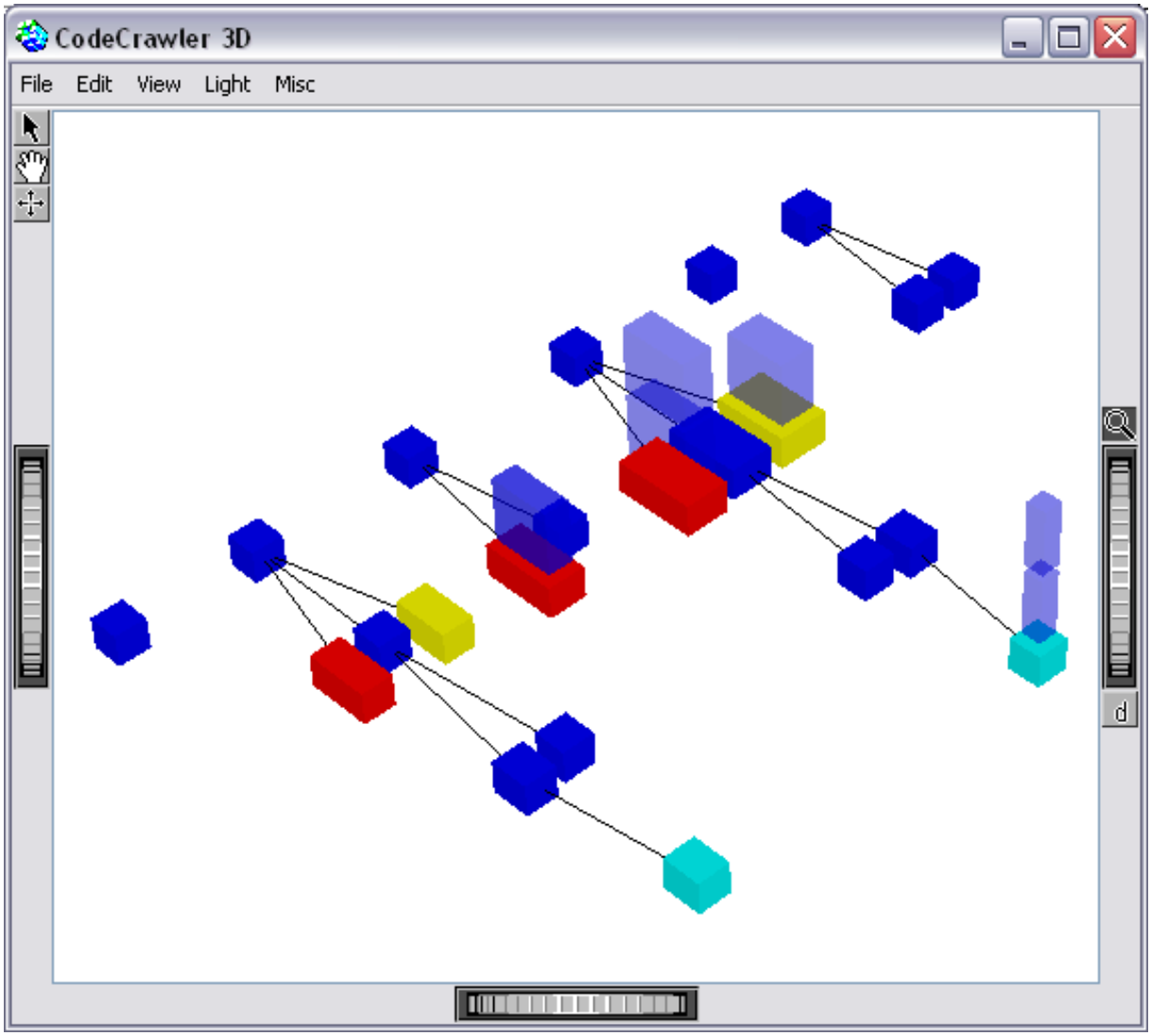

Figure 3.3: Initial position of the *dynamic feature interaction view* of a test model

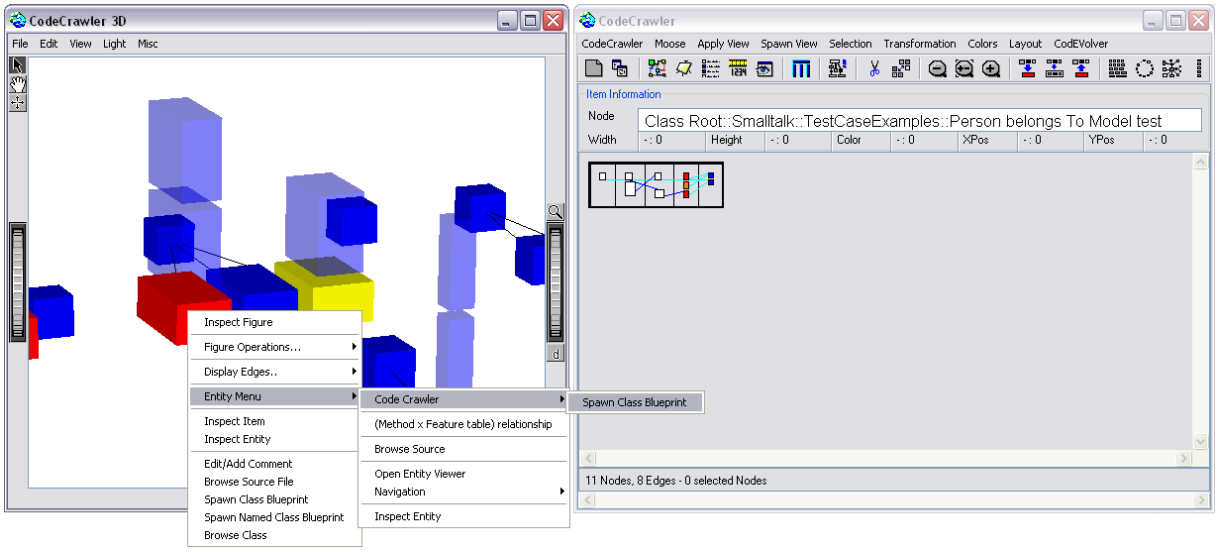

Figure 3.4: Zoomed in and rotated view with a opened *class blueprint*

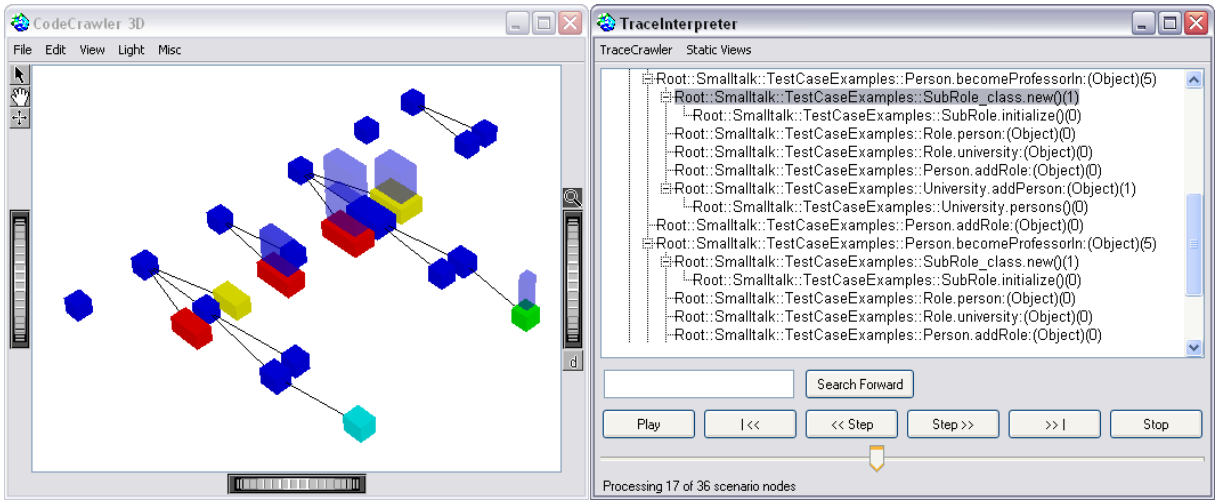

Figure 3.5: An instance creation within our test model

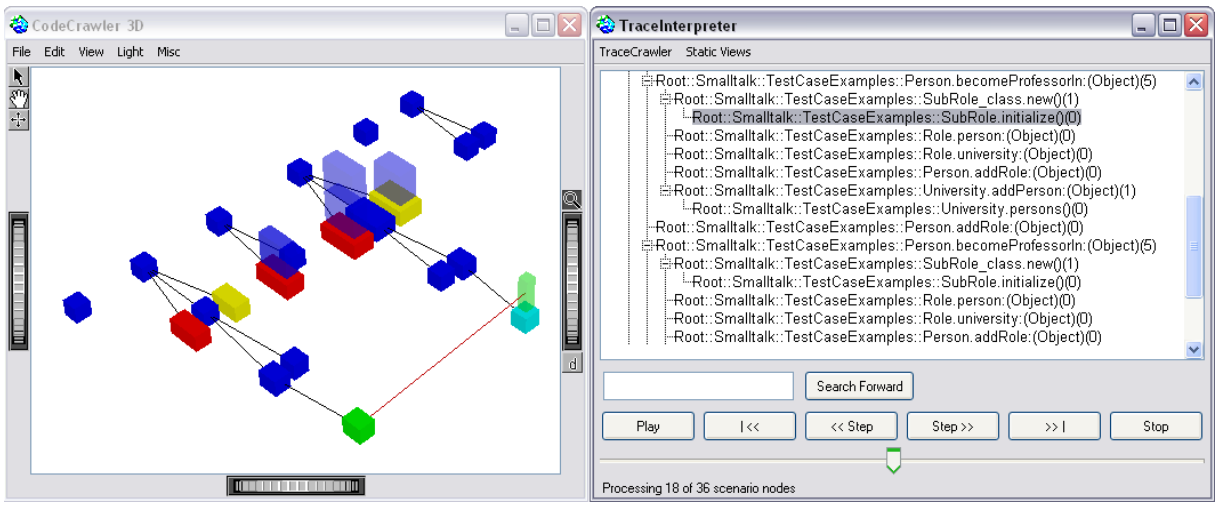

Figure 3.6: A message invocation from the meta-class to the newly created instance

Figure **??** shows the initial position at a specific point of a *feature-trace* of a test model using our *dynamic feature-trace view*. To determine to which class a specific tower of instance boxes belongs we provide context-sensitive menus as shown in Figure **??**. This is the same view but zoomed in to the point of interest. The menu in this case provides the possibility to get a more fine-grained view of the class, namely the *Class Blueprint* introduced by Lanza *et.al.* [**?**,]. This view reveals the internal structure of a class in terms of attributes and methods which are characterized in different groups.

3.1.3 Exemplification

In Figure **??**, Figure **??** and Figure **??** we show an example test model with four feature-traces interpreted by *TraceCrawler*. Our test model contains 22 classes and meta-classes and models people and their roles at the university.

TraceCrawler allows to visually step through the traces: At each point in time of the trace we see the current state of the trace and we can navigate backward and forward within the trace. On the right side of

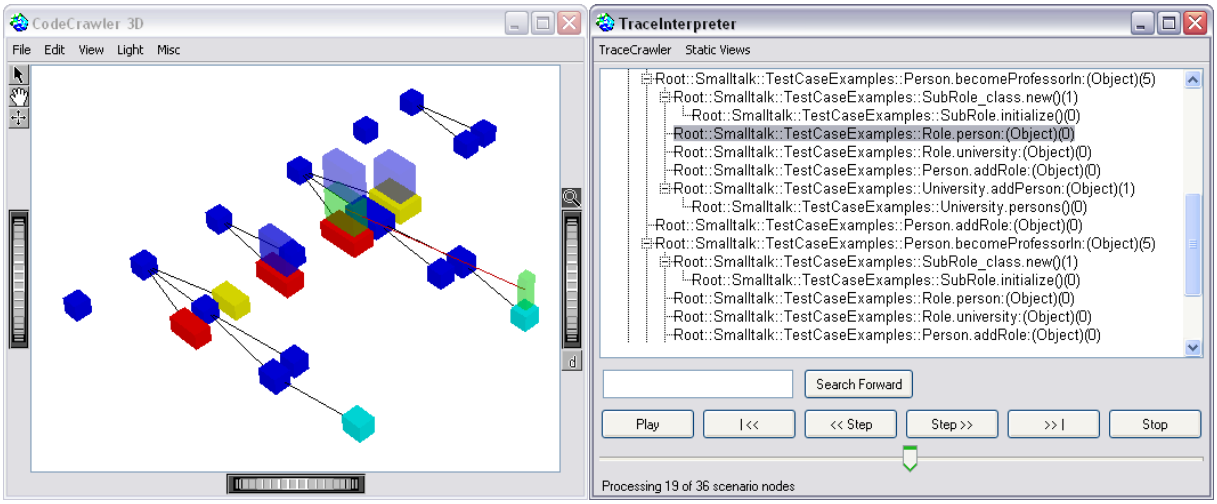

Figure 3.7: A communication between two instances of different classes

our example the feature-trace is shown as tree of method invocations with a highlighted node which is the currently active method call. On the left side *TraceCrawler* renders the interpreted data to the proposed 3-D visualization. Furthermore we allow searching to the next occurrence of a class and/or method and *TraceCrawler* navigates directly to it.

The three figures show three different steps during the trace:

1. Figure **??** shows a instantiation of a class (the creation of an object) of the class *SubRole*. The active class therefore is highlighted with a green color and a new instance box was drawn on top of it.

2. Afterwards the meta-class *SubRole_class* initializes the newly created instance. This message invocation is drawn as a red edge between the meta-class and the instance boxes which are both highlighted. This view is shown in Figure **??**.

3. In Figure **??** the same instance is communicating with another instance on the tower of the class *Person*. Again, the two communicating entities are highlighted with a green color and a red line is drawn between the two boxes.

This short scenario shows how one can understand the dynamic behavior of a software system. In our simple example the instance of *Person* is creating a *SubRole* and is setting the attribute *person* to itself using the appropriate method.

3.2 Feature Hotspot Analysis

We analyze the feature-traces to detect *feature hotspots*. We consider objects that appear as central points of communication to be *feature hotspots*. In other words they send and receive a higher than average number of messages than other instances. Such information is useful to understand the systems run-time behavior and allows us to answer questions such as the ones listed in the introduction.

The formation of a *feature hotspot* can be detected by the *dynamic feature-trace view* and an additional static view which we refer to as *instance collaboration view*. This view shows the same visualization at the end of the *feature-trace* to identify the relevant entities. Zooming to the relevant entities the software engineer can detect these *feature hotspots* and identify the parts of the feature that are producing it. Using

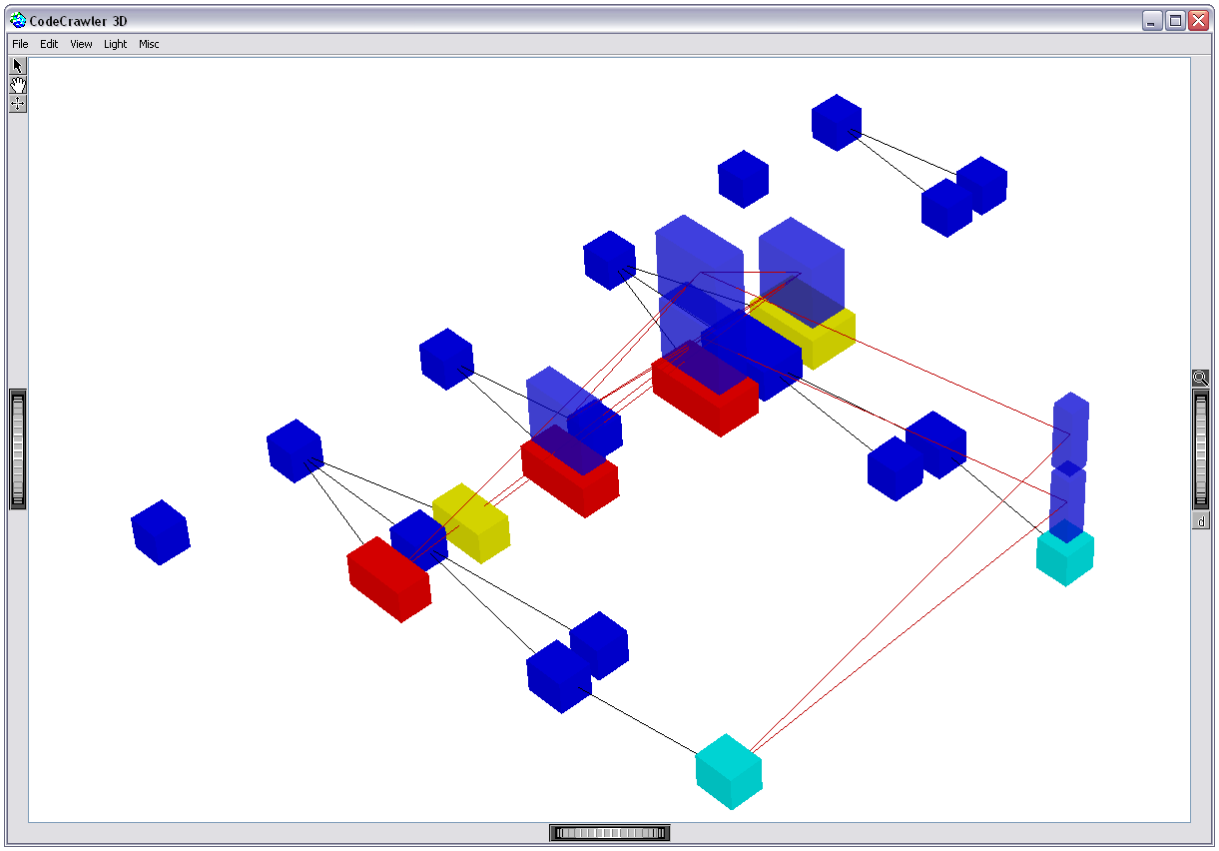

Figure 3.8: The static instance collaboration view of the test model

the user interface the software engineer reveals the method calls with their sender and receiver and may then change to the source code to analyze the hotspot further.

Using the test model presented above and a specific feature we show in Figure **??** an example of the *instance collaboration view*. The red lines are all method invocations analyzed during the specific feature-trace. Using this view one can recognize the entities communicating during the execution of the feature. Instances of the classes *Person* and *University* are communicating more than the other entities and are therefore *feature hotspots*.

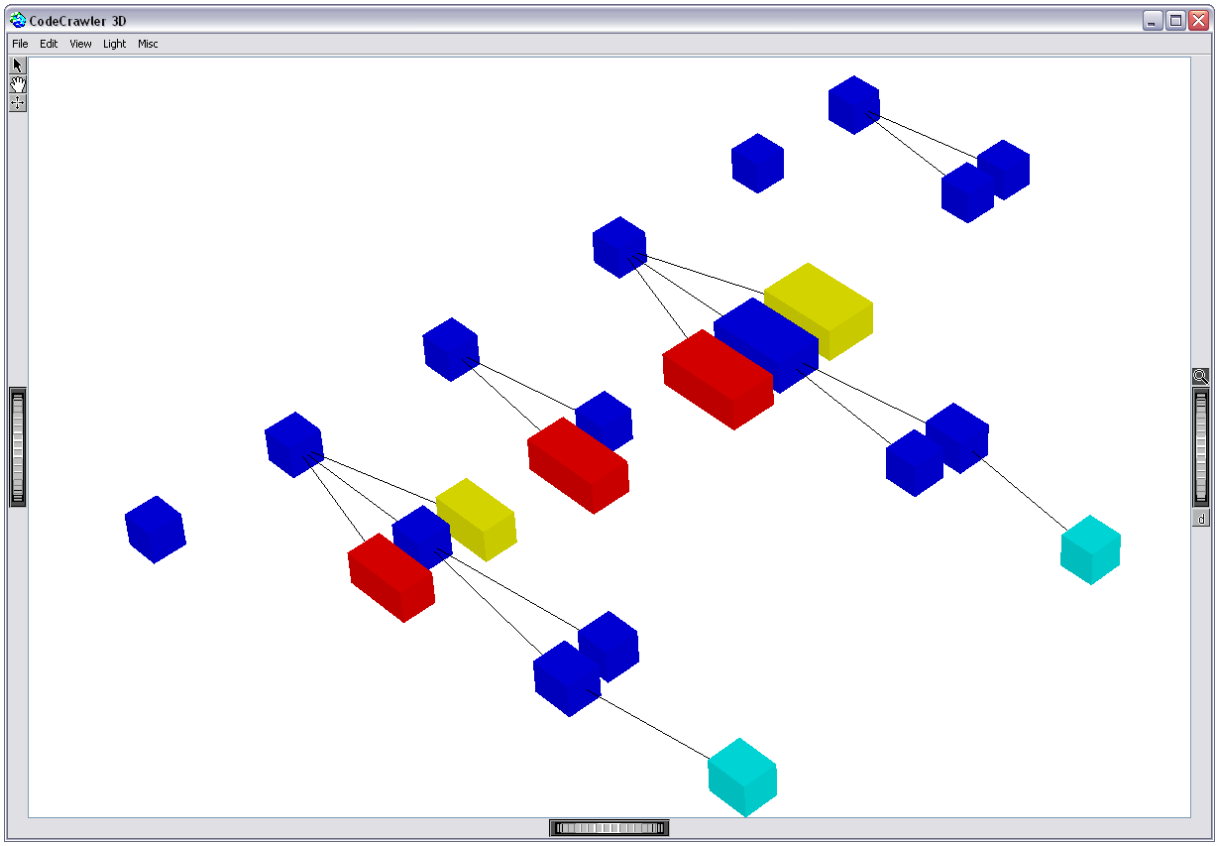

Figure 3.9: The static feature interaction view of a test model

3.3 Feature Interaction

To map features to source code we propose a static view which is based on a polymetric view. Its color metric is the FC measurement which measures for each class if it is active in one or more features or inactive at all. The result is the *static feature interaction view* which helps to identify parts of the software system which participate within one or more feature. Zooming into the visualization and using the context-sensitive menus the user is able to identify the class and to obtain more fine-grained information about the software entities of interest.

In Figure **??** the *static feature interaction view* is used to visualize a simple test model. The FC measurement is based on four features of our simple example system. Out of our view we detect

- 15 *Not Covered* (NC) classes and meta-classes
- 2 *Single-Feature* (SF) classes and meta-classes, namely *Professor_class* and *Professor*
- 2 *Group-Feature* (GF) classes and meta-classes, namely *University_class* and *University*
- 3 *Infrastructural* (I) classes and meta-classes, namely *Person_class*, *Person* and *PersonTest*

Chapter 4

Case Studies

4.1 Introduction

In this chapter we present the result of applying our visualization technique introduced in Chapter **??** to two software system:

1. SmallWiki is an object-oriented wiki implementation written in Smalltalk [**?**,]. It provides common wiki functionalities like adding and editing pages, user authentication, etc. In Section **??** we present the results of analyzing five different features such as the login process, editing a page, search the wiki and others.

2. The Moose reengineering environment is a language independent tool written in Smalltalk to support reengineering and reverse engineering [?]. It provides metrics, querying techniques, import and export functionality from and to different formats and a lot of tools which are based on it. In our case study we focus on the import and export of CDIF files and the loading of a model directly from Smalltalk code. See Section **??** for the presentation of the results.

4.2 Towards a Methodology

By applying our visualization to *feature-traces* of our case studies we developed a methodology to analyze software system based on *feature-traces*. Below we describe a step-by-step guide of our analysis procedure. Steps 2 to 4 are iteratively executed for each feature exercised on the software system.

1. The first step is to analyze the *static feature interaction view* which reveals the part of the systems which are active during one or more features. Using this view we get a first impression of the parts of the software system that are participating within the analyzed features. We gain an understanding of how feature interact which is an important information for the software maintenance. Moreover, we are able to reveal exceptional entities within the static structure of the software system. Exceptional entities may be classes with an exceptional form and that participate within one or more features.

2. As a second step we analyze the tree view of the *feature-trace* to get an coarse-grained overview. This view reveals the basic structure of the *feature-trace*.

3. Most of the information about the run-time behavior can be gained from the *instance collaboration view*. This view shows the instances that were created during the execution of a feature and the message invocations between them and their classes. It is the basis to discover *feature hotspots*. This view can be reduced to a specific part of a *feature-trace* and certain method invocations can be filtered out.

4. To understand how the *feature hotspots* were formed we look at the animation and step through the trace. This reveals the part of the *feature-trace* which leads to the hotspot. Switching to the source code using the context-sensitive menus of all class and meta-classes is the last step to analyze the run-time behavior more fine-grained.

4.3 Smallwiki Case Study

Our second case study is based on SmallWiki, an object-oriented wiki implementation written in Smalltalk [?]. The version we analyzed (1.297) consists of 288 classes. The following five features were analyzed:

1. Login Authentication (4008 message invocations)
2. Edit a Page (5608 message invocations)
3. Edit a Template (8435 message invocations)
4. Search the Wiki (7742 message invocations)
5. Show History of a Page (5563 message invocations)

As SmallWiki is written in Smalltalk we install the method wrappers of our tool *TraceScraper* to collect the *feature-trace*. Therefore the SmallWiki web server is started and the use cases are executed. We use WebUnit[1] to run our use cases.

4.3.1 Overall Overview

As a start we analyze the *static feature interaction view* which reveals the parts of SmallWiki which participate in all features, *i.e.* which are infrastructural and are colored red. Using the *Moose* reengineering environment and the FC measurement we detected 65 classes and meta-classes which are infrastructural. These can be seen in Figure **??** as red boxes. This represents the FC measurement value of 3 and we are using the colors of the heat metaphor presented in Figure **??**. Moreover, this view allows to identify exceptional entities. One class that stands out is the *HTMLWriteStream* class. This box is deeper than the others because the metric NOM, *i.e.* number of methods, which has a value of 87 which results in a deep box. We verify by scanning the source code that this class plays a key role as it is responsible for the generation of the HTML source code.

In the further sections we describe the five features in detail. We apply our filter so as not to visualize method invocations to instances from their own meta-classes. We use the *instance collaboration view* and zoom into the main inheritance hierarchy active during the execution of the features.

[1] http://webunit.sourceforge.net/

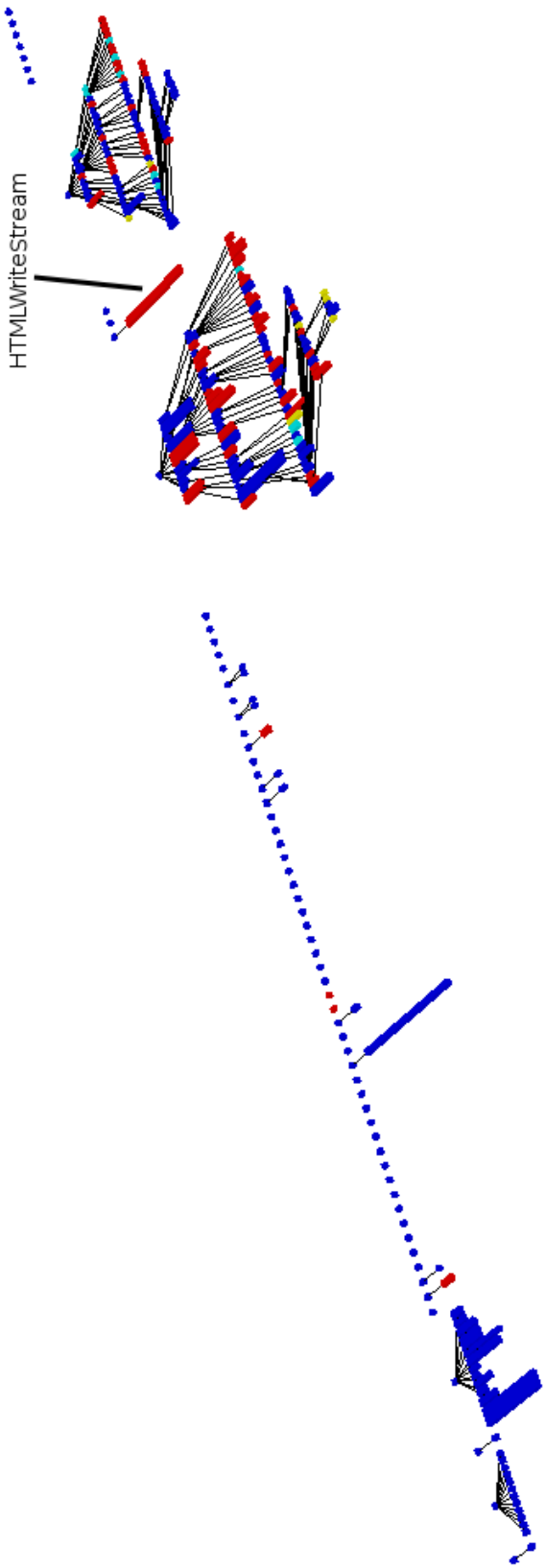

Figure 4.1: An overview of SmallWiki using the static feature interaction view

4.3.2 Feature 1: Login Authentication

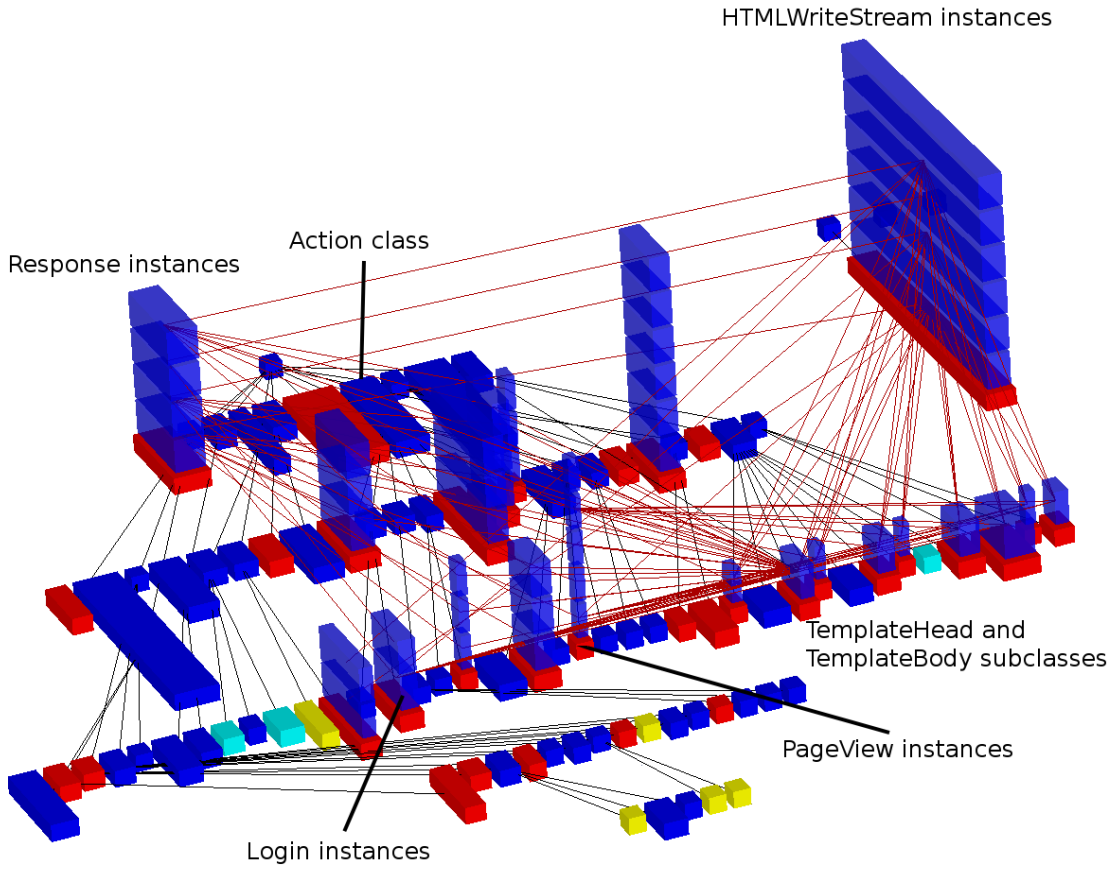

Figure 4.2: SmallWiki login feature

Looking at Figure **??** we detect four *feature hotspots*. Looking at their name and source code we can reveal most of the run-time behavior of the login scenario.

Login: This class is part of the *Action* class hierarchy. By executing the login feature it is no surprise that the instances of this class perform the task of login the user. One of its instances is heavily communicating with instances of the template hierarchy (subclasses of *TemplateBody* and *TemplateHead*). In SmallWiki templates are used for the composition of pages. That is why this is the instance that renders the login form and is executed to perform the login itself. The second instance is asked if the login is still correct while rendering the current page that is rendered after the login.

Response: The instances of this class are responsible for handling the HTTP response which is sent back to the browser. It is responsible for the storage of cookies, the response stream and HTTP functions such as redirection. During the login scenario four HTTP responses are sent back. We see this from the number of instances of this class. One of its instances is not requesting any information from the template hierarchy. This is caused by a HTTP redirection after the login form was sent back from the browser which we can see by running the animation of the login scenario.

HTMLWriteStream: As we revealed in the previous section the instances of this class are responsible for the generation of the HTML code.

PageView: This class is also part of the *Action* class hierarchy and is responsible for the rendering of the page. Due to the fact that pages are composed of the templates it is also heavily communicating with instances of this class hierarchy.

4.3.3 Feature 2: Edit a Page

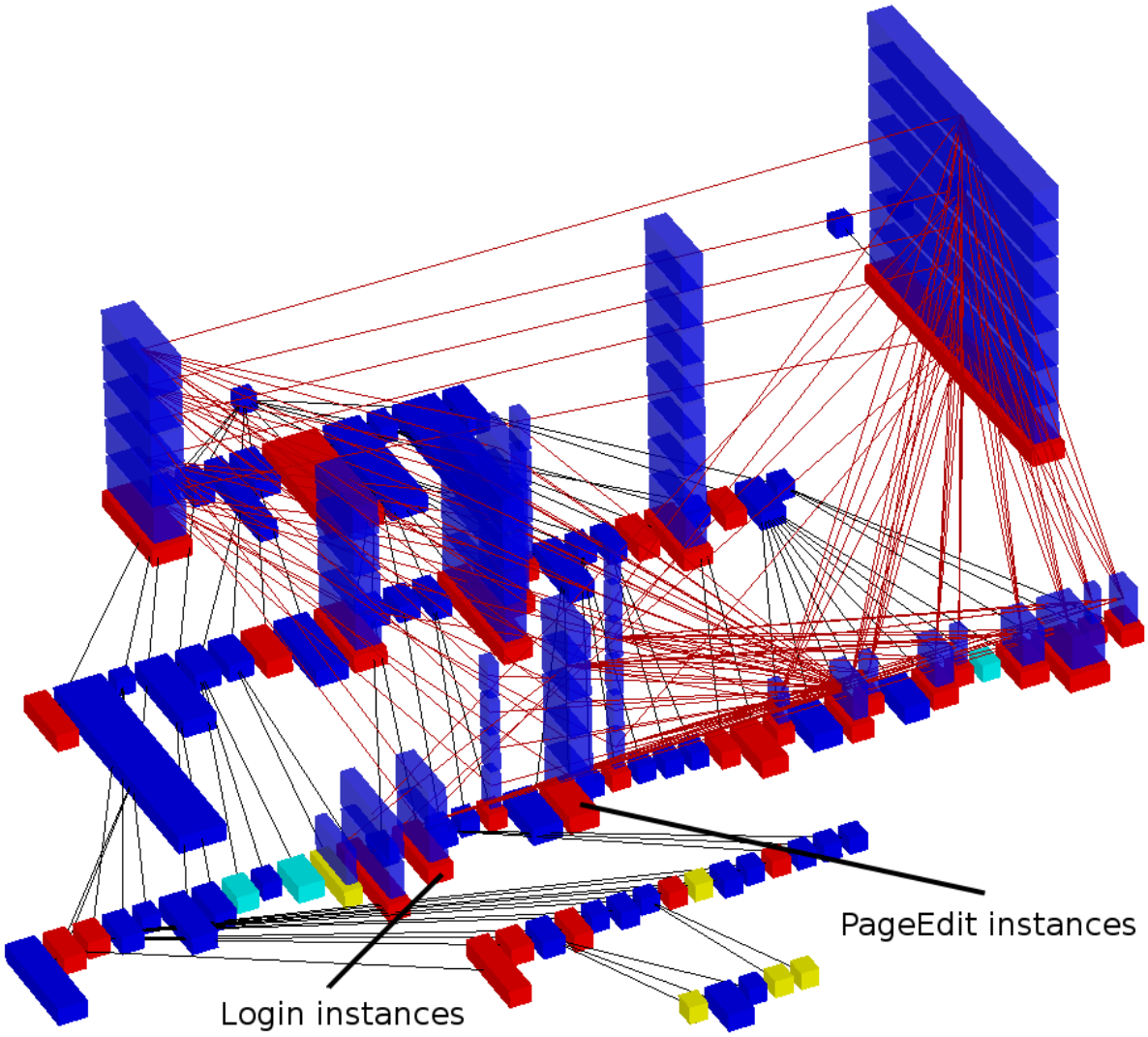

Figure 4.3: SmallWiki edit page feature

This feature allows the user to edit the content of a wiki page. This feature is secure, in other words a login has to take place before. Comparing Figure **??** and Figure **??** we detect that the differences between the scenarios are small. In Figure **??** we see that the there are again *Login* instances. Looking at the use cases that are used to generate the *feature-trace* we reveal as expected that the login scenario is part of the edit page scenario. Apart from this important fact we also detect one new *feature hotspot* which is not part of the login scenario:

EditPage: There is one instance of the tower which is communicating heavily with the template class hierarchy. This instance renders the form to edit the wiki page and saves the submitted content to the model.

4.3.4 Feature 3: Edit a Template

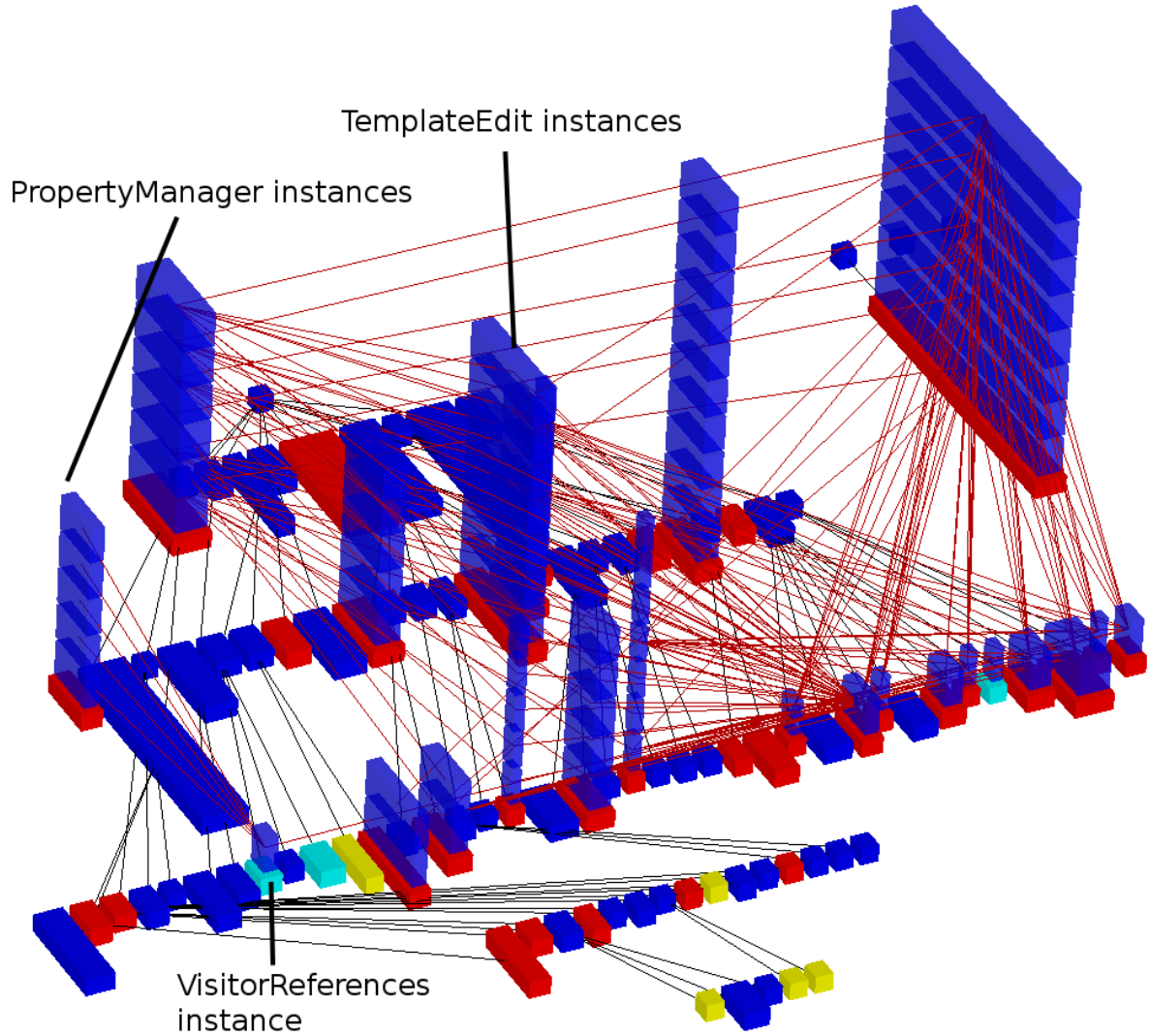

Figure 4.4: SmallWiki edit template feature

This feature allows a user to modify the look and feel of pages by changing a template which affects the position, color, etc. of the page elements. We see that this scenario looks quite complicated compared to the previous introduced ones. A visually striking element in Figure **??** is the tower of instances of *PropertyManager* that all communicate with an instance of *VisitorReferences* which is a single-feature class. This implies that while editing a template the properties of the template change and this is recorded somewhere. Apart from the *feature hotspots* we found in the first feature we detect one new *feature hotspot*:

TemplateEdit: This class is part of the *Action* hierarchy and provides the functionality to change a template using various commands. Therefore it is no surprise that instances of this class are heavily communicating with the subclasses of the template hierarchy.

4.3.5 Feature 4: Search the Wiki

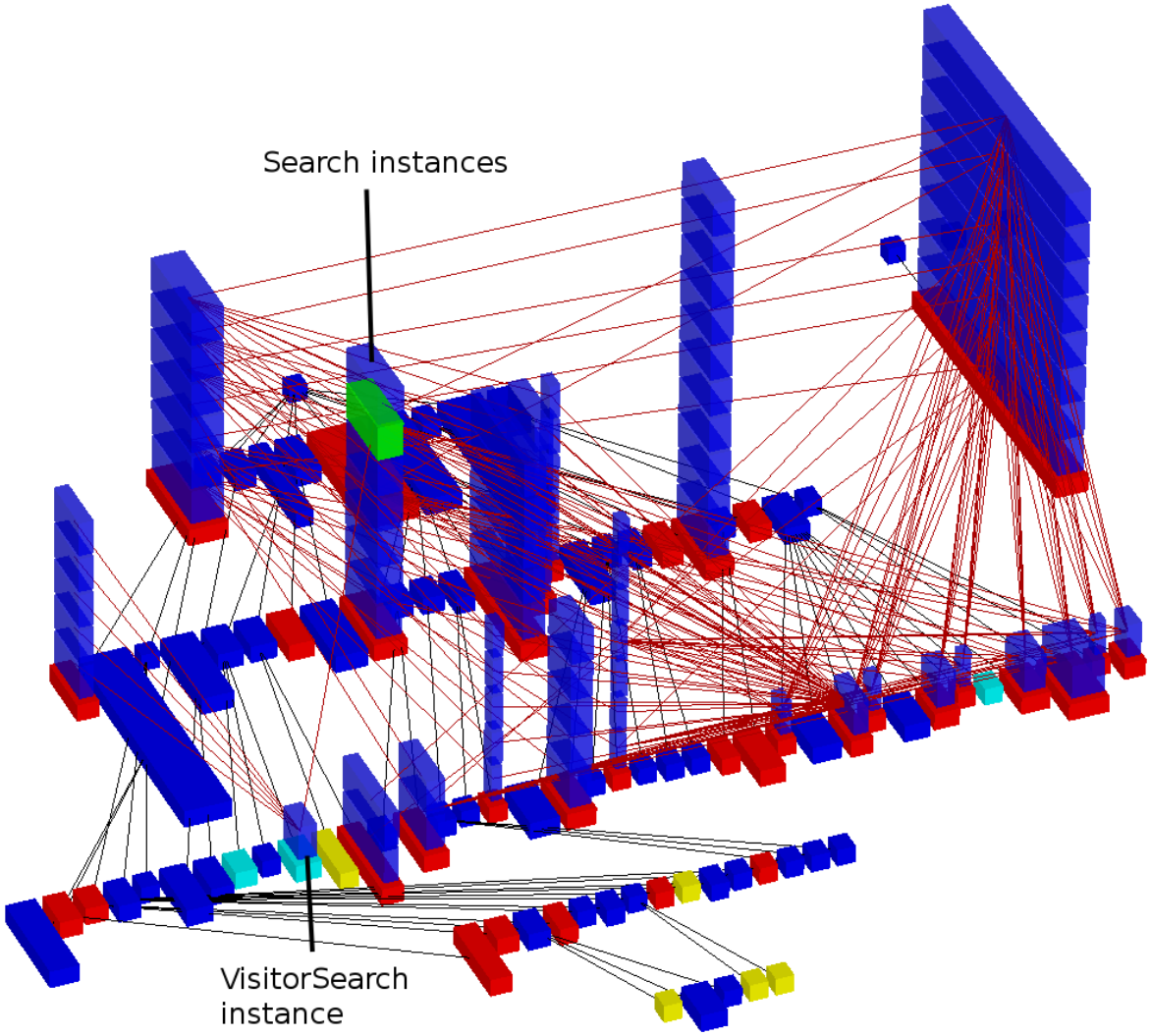

Figure 4.5: SmallWiki search feature

This feature allows the user to search for a specific string within the wiki. In Figure **??** we detect another singe-feature class, namely *VisitorSearch*. Already the name reveals that the search functionality of SmallWiki was realized using the Visitor pattern [**?**,]. Unfortunately we realize that the process of visiting the model of SmallWiki cannot be visualized. The reason for this is lack of information in the feature-trace about the SmallWiki model itself. That is why we do not have the instances which are part of the model within our visualization. We discuss this limitation in Section **??**. Nevertheless we detect another *feature hotspot* which is active during the search feature:

Search: This class is again part of the *Action* class hierarchy. One instance is communicating heavily with the template subclass instances. This is caused by the rendering of the results the search visitor returns.

4.3.6 Feature 5: Show History of a Page

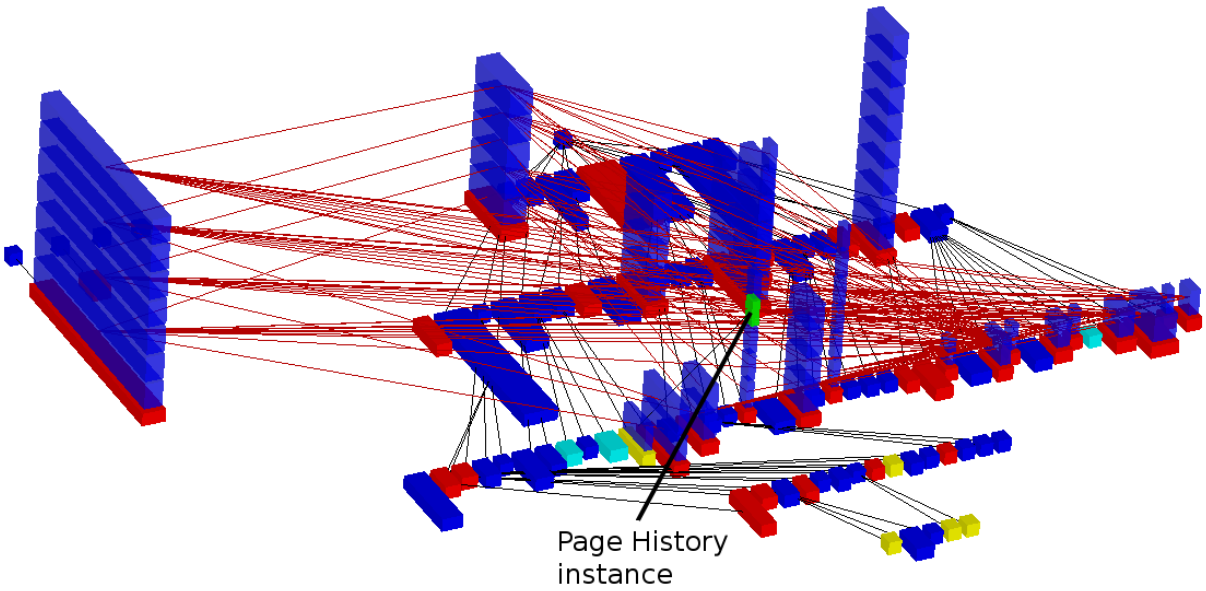

Figure 4.6: SmallWiki show history feature

This feature in SmallWiki allows the user to see a list of all pages in SmallWiki which have recently changed. The history features refers to the versions saved in the model of each page. This information is not part of the feature-trace and therefore not visible. Moreover, we detect one new *feature hotspot*:

History: One instance of this action is communicating with the template subclasses. This is caused by the rendering process of the version table which can be revealed by scanning the source code.

4.3.7 Discussion

Using our visualizations we were able to easily determine the parts of the software system that are participating in a feature. Comparing the different views we were able to detect *feature hotspots* that arise only in one specific feature and is therefore interesting to analyze further using other methods such as the *Class Blueprint* or source code reading. An important fact that was revealed using the visualization is that the login feature is part of all the other features. This was even revealed without looking at the use cases that were used to generate the *feature-trace*.

We also realized that analyzing more complex feature is difficult as there are a lot of messages being visualized. We detected that the rendering process is taking place in all features. Therefore a filtering mechanism which would filter these method invocations would be an easy way to realize simpler views. Such views would allow to focus even more detailed on the most interesting parts of the features.

A speciality of this case study is that SmallWiki is based on HTTP request and response. Action providers such as the search, edit page and others are created for each rendering of a page individually. Using our technique we can determine which instance is really executing the action using our *feature hotspots*. This is a significant support for the software engineer to understand the run-time behavior of the different features in SmallWiki. On the other hand because of the missing model information an important part of the feature-trace is not visualized. The reason for this limitation is that the creation of the model of Smallwiki is not part of the analyzed *feature-traces*. This leads to an incomplete view which does not reveal how the actions perform on the model.

4.4 Moose Case Study

This case study is based on the Moose reengineering environment [?] which provides also the meta-model for our visualizations. The version (3.0.25) we analyze consists of 792 classes and meta-classes. We are analyzing three features:

1. Loading a model of a software system from Smalltalk (71280 message invocations)
2. Exporting a model to a CDIF file (44836 message invocations)
3. Importing a model from a CDIF file from the file system (95040 message invocations)

The CDIF standard for information exchange [**?**,] which is mentioned within the feature names is used to exchange language independent model information between reengineering tools [**?**,].

4.4.1 Overall Overview

To analyze how the features interact with each other we start our analysis by looking for the *static feature interaction view*. This reveals the classes which participate in one or more features. In Figure **??** we see that most of the infrastructural classes are residing in the *Model* and *Model_class* hierarchy. These boxes are colored in red in the visualization. Especially remarkable using this view is that there are no group-feature at all and only a small number of singe-feature classes. This indicates that there is reuse of code. We expect this as the features are providing similar functionality.

In Figure **??** we zoomed in to the *Model* hierarchy and are looking to exceptional entities. We are looking for infrastructural classes that are colored red and which have a special form. The most eye-catching class we see is *MSEModel*. Switching to the source code using the Smalltalk System Browser we reveal that this class is responsible for the storage of the meta-model and is providing various functions on it. This class is very tall because of the metric NOA = 0 (width) and NOM = 188 (depth). Another class that stands out in this view is *FAMIXClass*. This class is part of the FAMIX meta-model which is no surprise as our features are concerned with loading and exporting FAMIX models. Besides we detect the *ImportingContext* as a further exceptional entity. This class controls the importing process. Another thing this view reveals is that the whole *Operator* subhierarchy is used in all features. *Operators* are responsible of computing metrics or other properties based on the meta-model. It seems that those are computed within each feature. Scanning the source code we are able to confirm this assumption.

In the following sections we describe each feature in detail and then discuss the results. We apply our filter so that message calls from instances from its own meta-class are not visualized. We use the *instance collaboration view* and the tree view of the *feature-trace* to show the runtime behavior of the presented features.

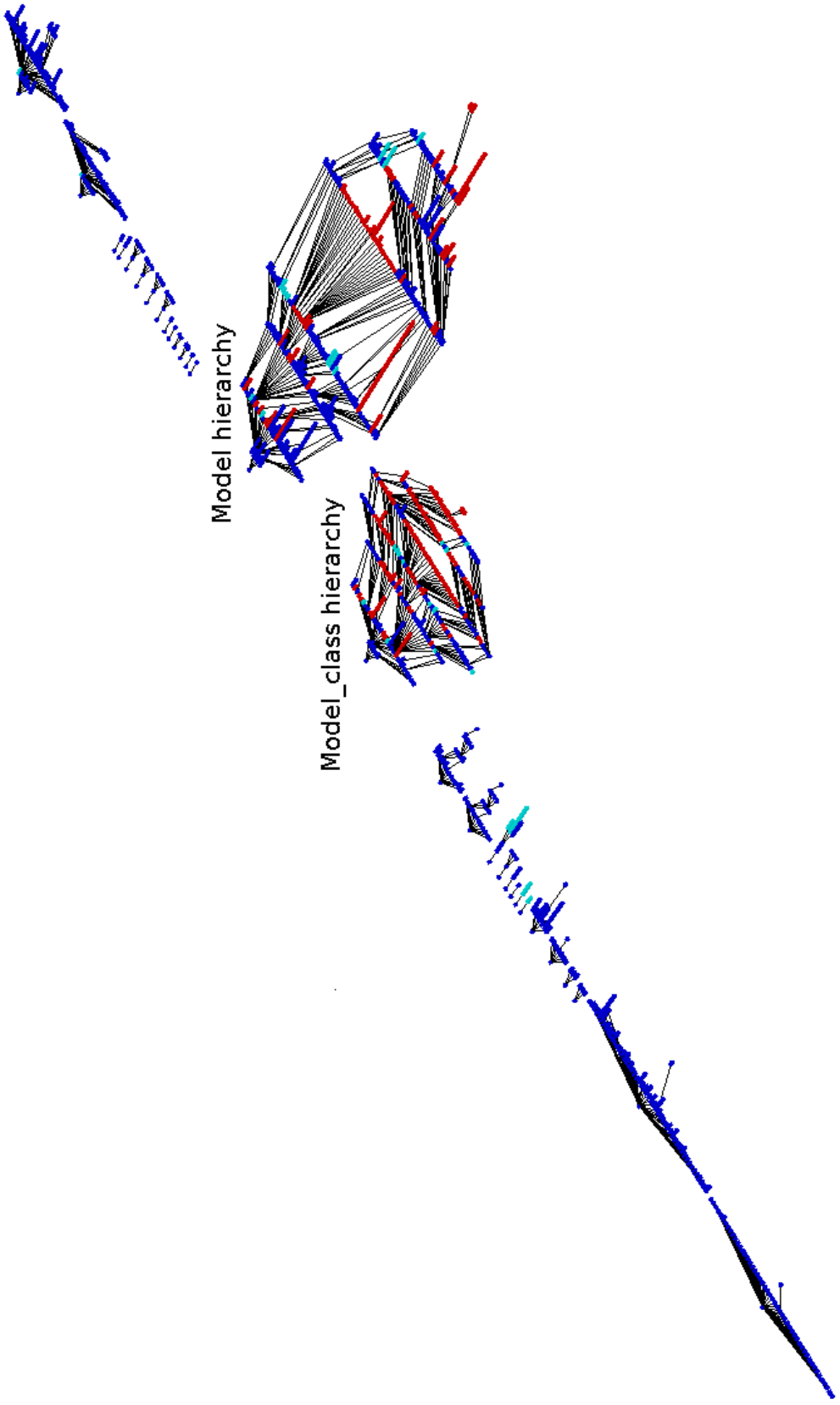

Figure 4.7: An overview of Moose using the static feature interaction view

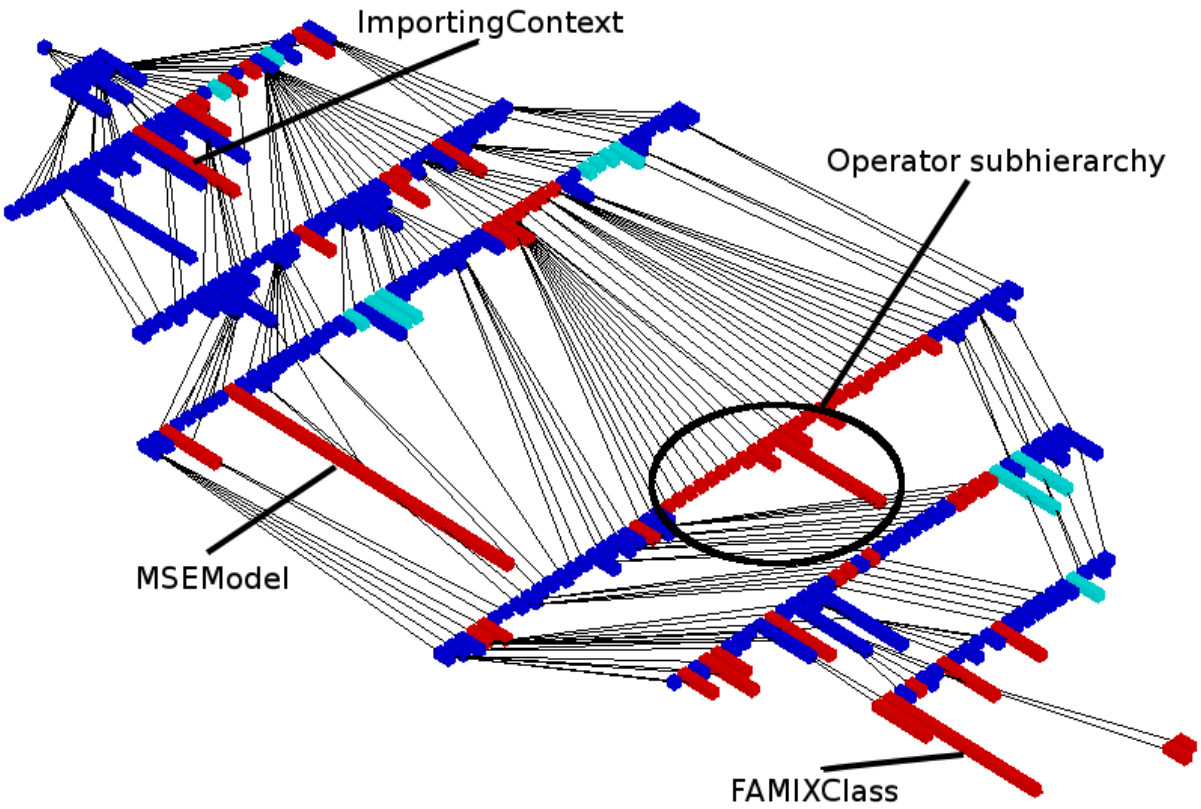

Figure 4.8: A view of the Moose model hierarchy using the static feature interaction view

4.4.2 Feature 1: Loading a Model from Smalltalk

Overview

To start the interpretation of this feature we look at the *feature-trace* rendered as a tree in Figure **??**. Looking at the first levels we reveal that the *feature-trace* consists of two parts. A setup routine which is communicating with the *ModelManager*, *EntityTypeManager* and *AbstractEntity*. We assume that this part of the *feature-trace* prepares *Moose* to import the model. The second part of the trace seems to be concerned with the loading of the model itself. Our analysis reveals that the *VisualWorksImporterFacade* which receives a message *#doImport* and assume that this message is responsible to load the model from Smalltalk.

To prove our assumptions we filter our *feature-trace* to those parts and start our *static instance collaboration view*.

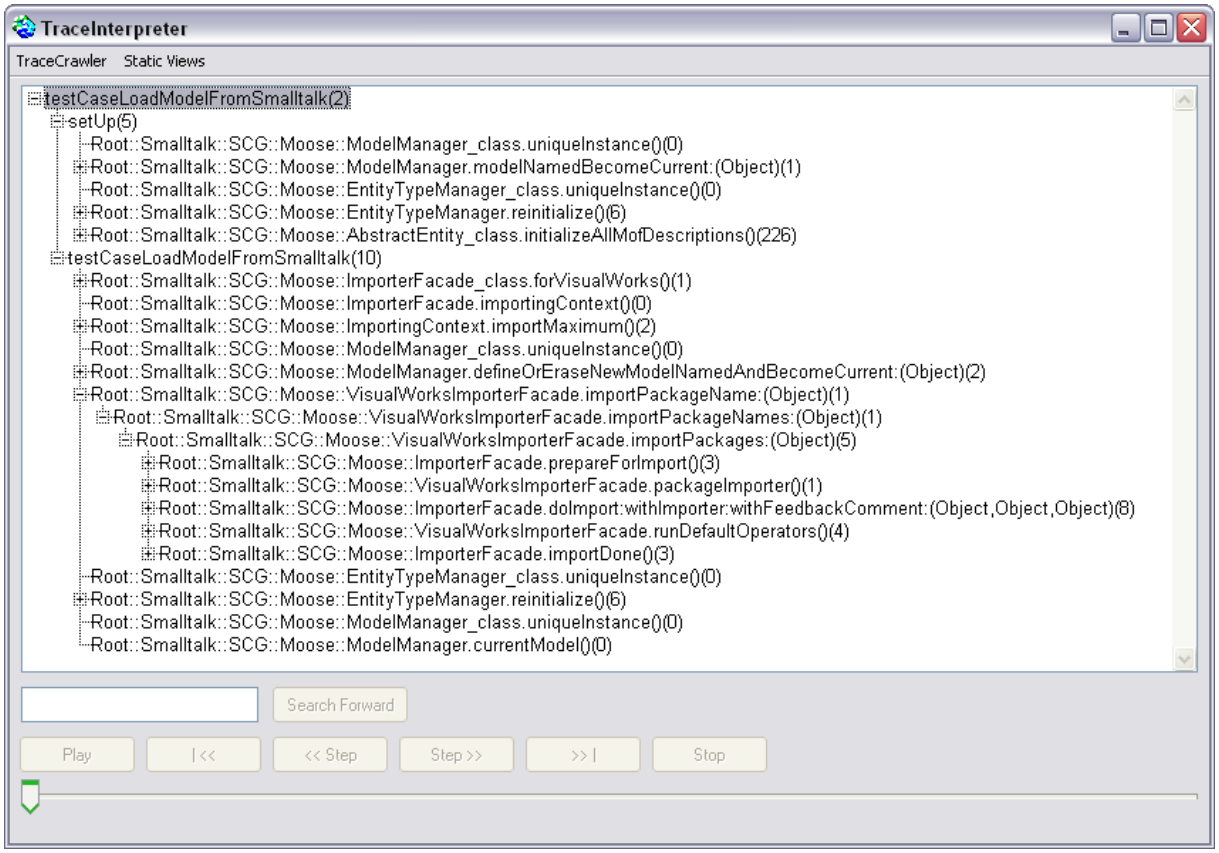

Figure 4.9: The tree view of the load model feature

Setup

To analyze this part of the *feature-trace* we present three screenshots. In Figure **??** we show the *Model* and *Model_class* hierarchies which are the only ones participating in this part of the feature. At a distance we detect several high towers which are caused by a high amount of instances for a specific class. *MOFExtendedClass*, *MOFExtendedMultipleValueAttribute*, *MOFExtendedAttribute*, *MOFPackage*, *MOFImport* and *EntityType* are their class names which we collect using the *Entity Inspector*. Moreover, we see several *feature hotspots* which we analyze now in more detail.

First we focus on the *Model_class* hierarchy as shown in Figure **??** which reveals several *feature hotspots*:

- *MetaModelRepository_class*: The class comment in the source code shows that this class is used as a repository of model descriptions. This meta-class is mostly used to access its unique instance which reveals that the Singleton pattern [?] is used. Unfortunately we are not able to visualize this instance as the creation is not part of the analyzed *feature-trace*.
- *AbstractEntity_class*: The messages this meta-class sends to other meta-class is *#registerMofPackage*. We assume that this is used to initialize the model descriptions of the entity types. Using the tree view of the *feature-trace* we detect that this action is closly related to the *MetaModelRepository_class* above which is the repository of those descriptions.
- *EntityTypeManager_class*: This class is accessed by all the *EntityType* instances. Again the messages reveal that a Singleton pattern is used and that the creation of the unique instance is not part of the *feature-trace*. Nevertheless, the source code implies that the created entity types are registering themselves within the *EntityTypeManager*.

We switch our view and focus on the *Model* hierarchy in Figure **??**. We see some action going on

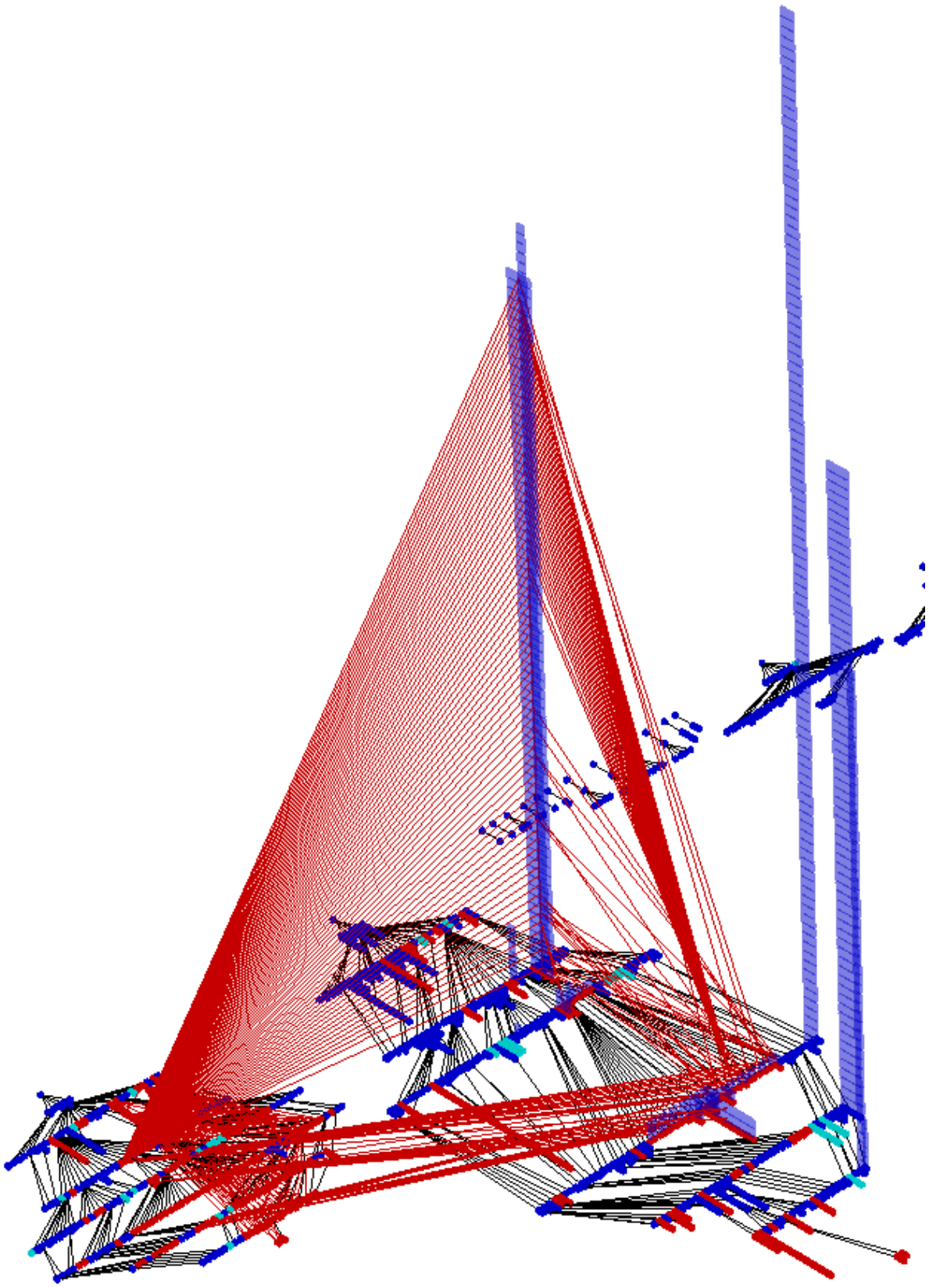

Figure 4.10: An overview of the setup routine of the Moose load model feature

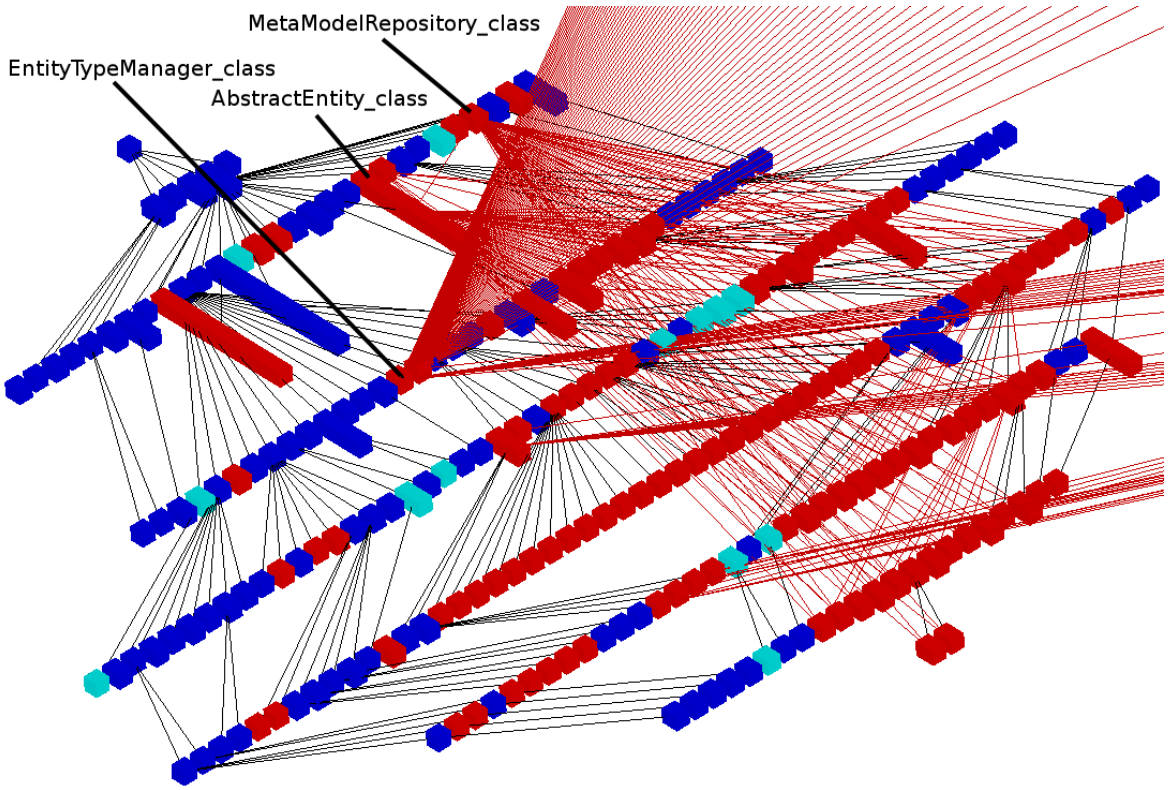

Figure 4.11: The *Model_class* hierarchy of the setup routine in the Moose load model feature

within the *PropertyOperator* and *AbstractFactory* subhierarchy but only a single *feature hotspot* located at the *CCEntityTypeFactory*. This class is added by *CodeCrawler* to this subhierarchy and seems to add necessary entity types to *Moose*.

If we analyze all the collected information above together with the tree view we are now able to infer the intent of this part of the *feature-trace*. Basically we detect two main purposes:

1. Registering Entity Types:
   Started by the unique instance of *EntityTypeManager* all known entity types are initialized and registered. Moreover, for all entity types the appropriate operators, menus, expressions, etc. are initialized. The source code reveals that there is an extensible environment to add new entity types, operators, etc. by subclassing the appropriate factory class. This is for instance realized by *CCEntityTypeFactory* of CodeCrawler.

2. Initializing MOF descriptions:
   After registering all entity types the meta-description mechanism is initialized. In Moose each entity has its own meta-description which is provided by its class. These are added to the *MetaModelRepository* which manages all meta-descriptions for the models. This also leads to the high towers of instances of MOF[2] classes.

[2]Meta-Object Facility, refer to http://www.omg.org/technology/documents/formal/mof.htm

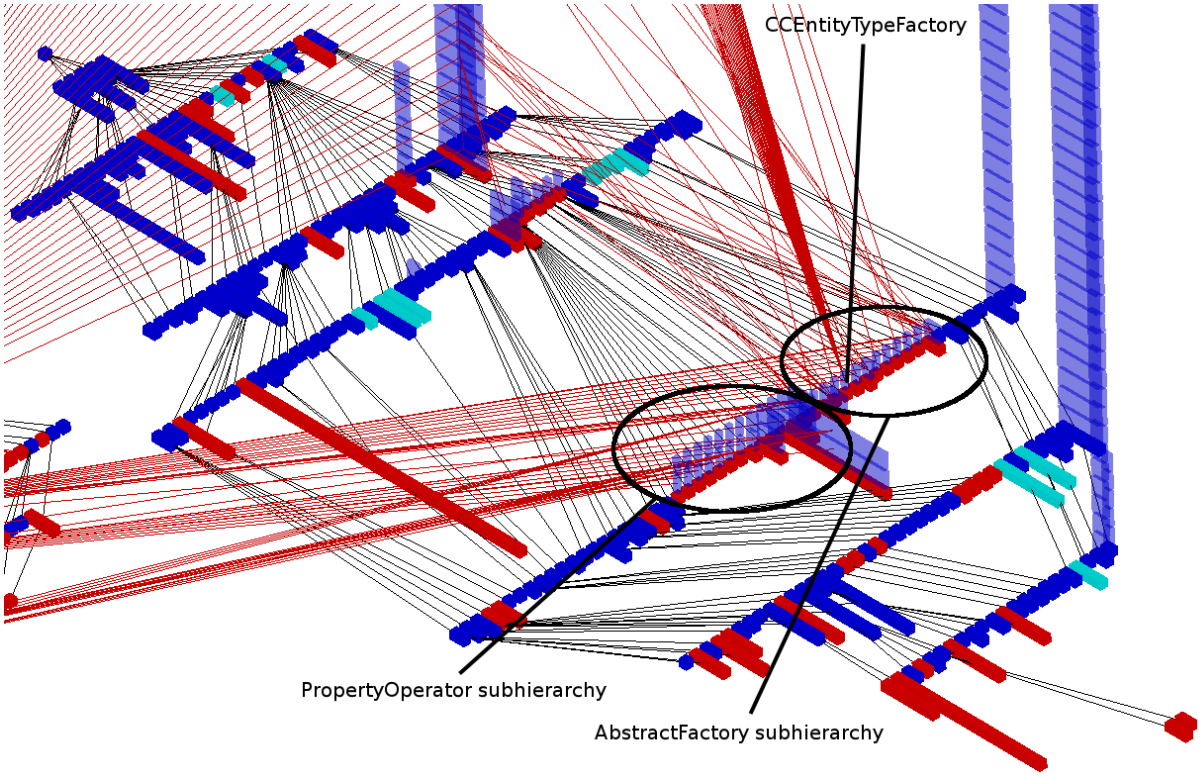

Figure 4.12: The *Model* hierarchy of the setup routine in the Moose load model feature

Import Package

The analysis of this part of *feature-trace* turned out to be really challenging due to the amount of messages that were drawn on screen. Especially messages between the *Model* and *Model_class* hierarchies tend to hide whole parts of the inheritance hierarchy. To overcome this problem we introduced another filter which allows to remove certain messages from the visualization. In this case we decided to remove the *#uniqueInstance* message invocations used by the Singleton pattern of several classes within the analyzed software system. Nevertheless the amount of messages (40'736) processed is very high and therefore the visualization is difficult to interpret.
In Figure **??** we see the *Model_class* hierarchy. We detect several *feature hotspots* which can be grouped together according to their inheritance relationship:

- *FAMIXFormalParameter_class*, *FAMIXLocalVariable_class*, *FAMIXGlobalVariable_class* and *FAMIXMethod_class*: These classes are subclasses of *FAMIXModelRoot_class* which provides various functions to create unique names for the different entities. These are the hotspots we see in Figure **??**.
- *FAMIXNameResolver_class*: This class provides a method to produce a signature for methods which is used by *FAMIXMethod* instances. Surprisingly we detect also functions to create unique names for different FAMIX entities which seems to be a code duplication although the messages are not used within this feature.

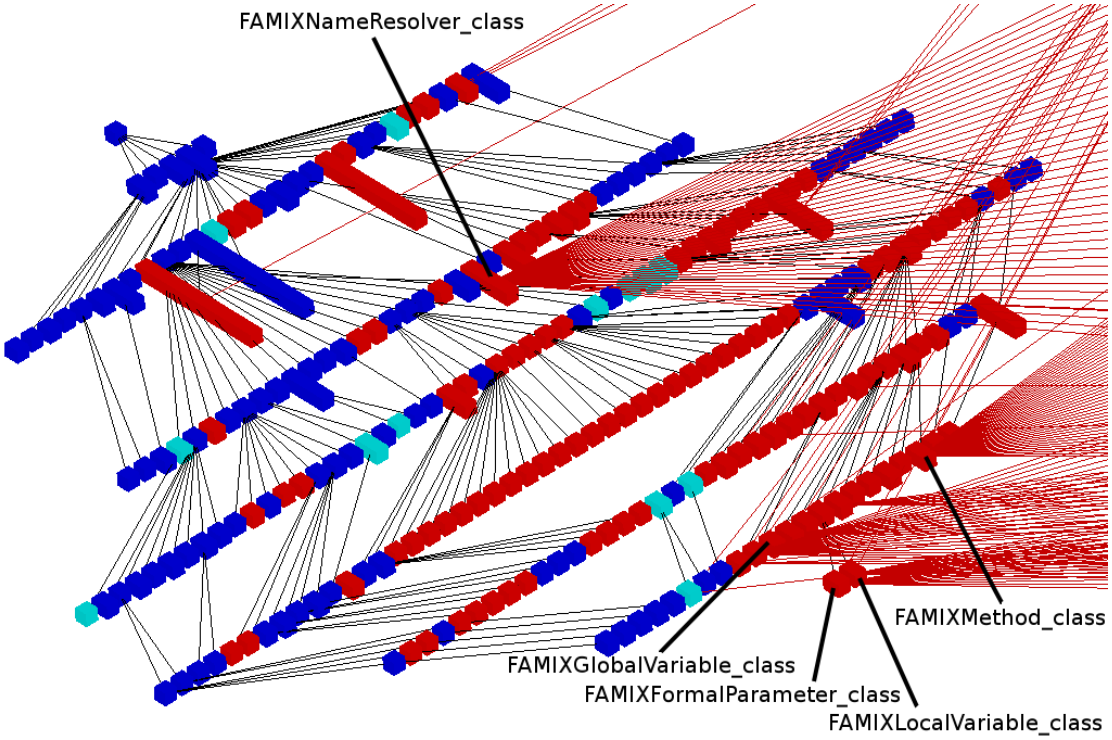

Figure 4.13: The *Model_class* hierarchy of the package import in the Moose load model feature

Figure **??** shows the difficulties of producing a screenshot of our *instance collaboration view* with a lot of method invocations. Nevertheless we are able to detect some *feature hotspots* and high towers of instances. We focus on these entities and try to identify their purpose looking at the source code.

- *ImportingContext*, *VisualWorksImporterFacade* and *VisualWorksPackageImporter*: The instances of these classes are responsible of importing Smalltalk code into a *Moose* model. Besides the *ImportingContext* the classes are single-feature classes which implies that they are only participating within this feature. We also reveal that the import is implemented using a Facade pattern[3].
- *ModelManager*: The instance of this class is a large *feature hotspot* which is no surprise as it is responsible of managing different models in *Moose*. It communicates heavily with all the instances of the FAMIX model.
- *MSEModel*, *MSEEnumeratedGroup*: While importing the Smalltalk package a *MSEModel* was created which holds the FAMIX model of the package. Enumerated groups are used to hold entities of the same entity type and are therefore also heavily used.
- *FAMIXClass*, *FAMIXMethod* and others: Compared to Figure **??** the view reveals that the FAMIX model was initialized and therefore several high instance towers appear. We count for example 22 instances of *FAMIXClass* which represents exactly the number of classes and meta-classes of the loaded package.

[3]Refer to http://en.wikipedia.org/wiki/Facade_pattern

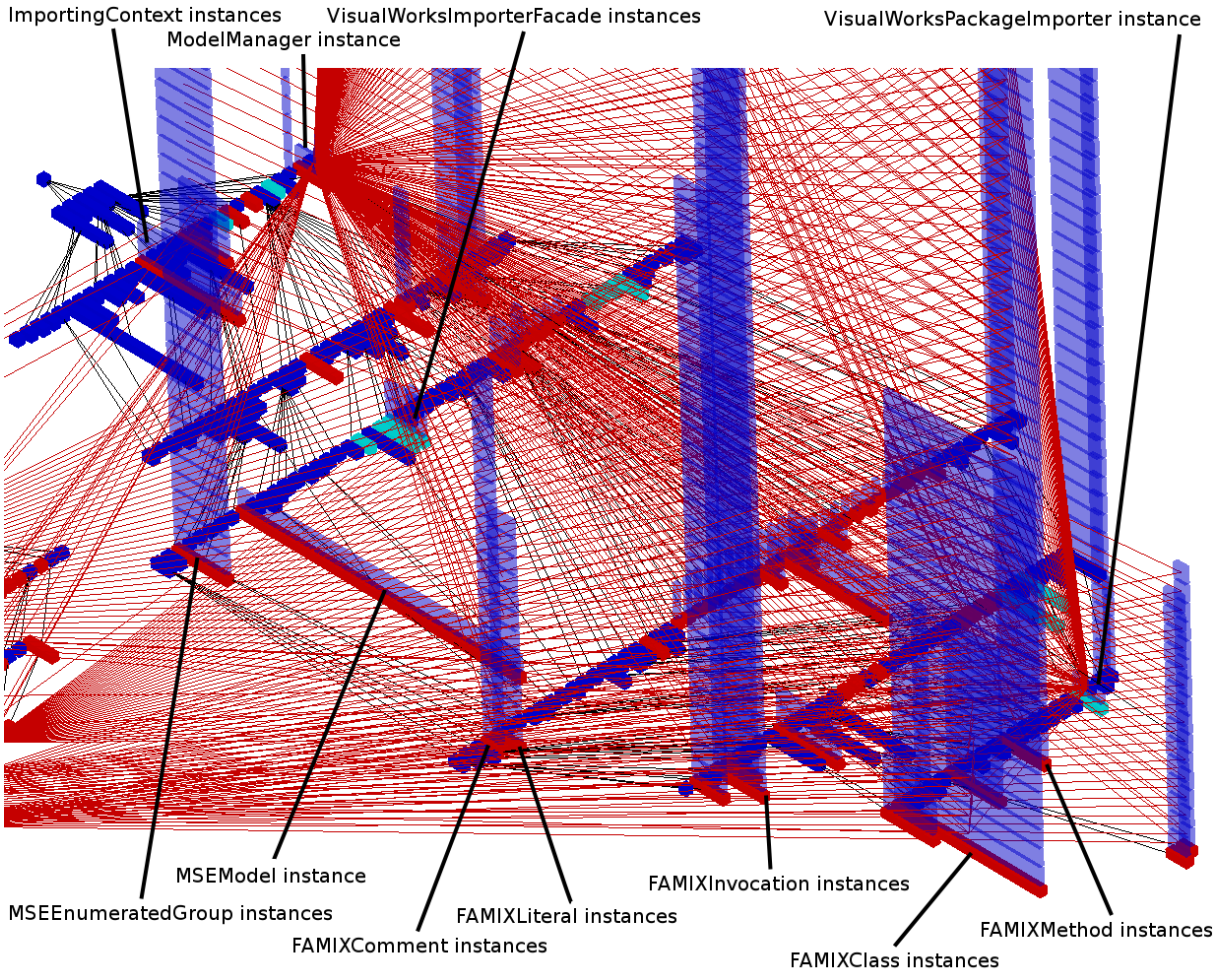

Figure 4.14: The *Model* hierarchy of the package import in the Moose load model feature

4.4.3 Feature 2: Exporting a Model to CDIF

Looking at the tree view of this feature in Figure **??** we detect that the setup routine is the same as in the feature presented before. We are able to prove this assumption by looking at the implementation of the test case which controls the trace extraction. In fact, the setup routine is called for each feature we analyze in this case study.

Moreover, this feature reveals one of the most important limitations of our approach. Figure **??** shows the *instance collaboration view* of the part of the *feature-trace* that seems to be responsible for the export of the *Moose* model to a CDIF file. We assume that the message *#saveCurrentModelOnCDIFFileNamed:* is starting the export. Looking at the figure we are surprised that besides the *towers of instances* only a small amount of message invocation is shown. The reason for this behavior was already detected during the Smallwiki case study. The creation of the model in *Moose* which then should be exported is not part of the analyzed *feature-trace*. That is why the identification of the instances is not possible and as a result no message invocation edges are drawn. To overcome such difficulties we would need to analyze as well as the execution trace, the software system state before the execution trace was started.

Nevertheless we are at least able to identify one important entity within the visualization. It is the instance of *CDIFSaver* which is responsible for the CDIF file writer. It is a single-feature class because it participates only in this feature.

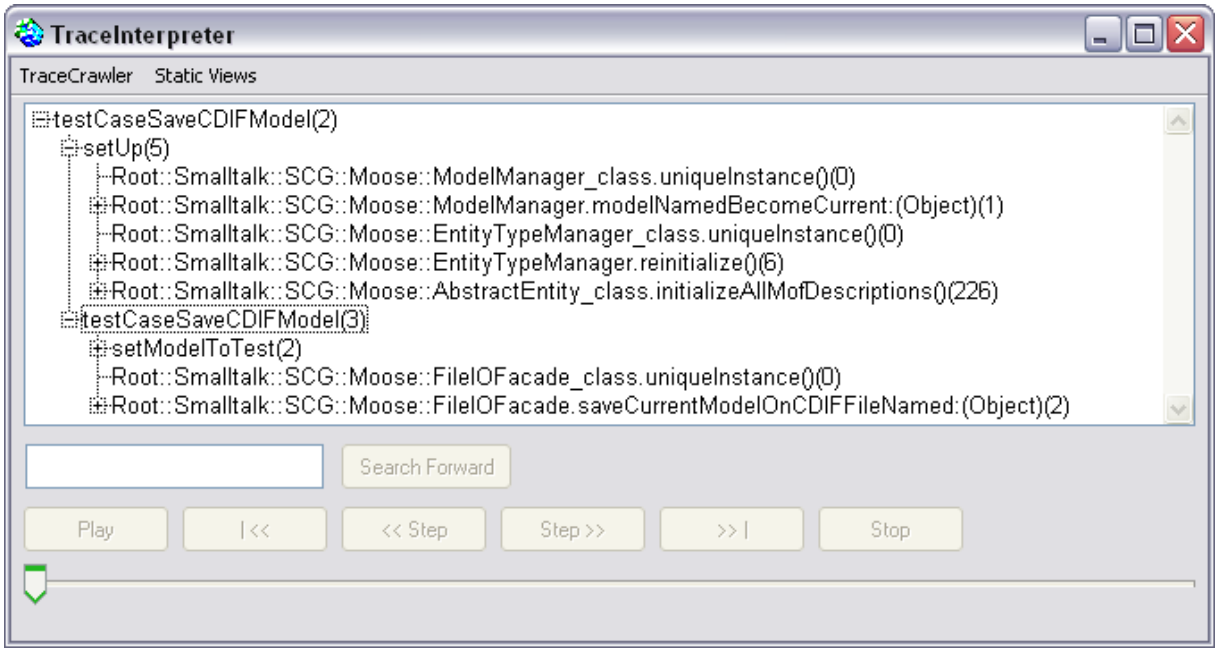

Figure 4.15: The tree view of the export to CDIF feature

4.4.4 Feature 3: Importing a Model from CDIF

As we proved in the section before the setup routine is the same for all feature and already analyzed. Therefore we focus on the main behavior of this feature. We start the analysis by looking at Figure **??** which reveals that the method *#loadModelFromCDIFFileNamed:* starts the import of the CDIF file. Visualizing this part of the *feature-trace* and zooming to the *Model* hierarchy leads to the view presented in Figure **??**. As a first impression we detect that the difference to the first feature presented in this case study seems to be small. We detect the following similarities and differences:

- *ImportingContext* and *CDIFImporter*: We detect that the *ImportingContext* is again used. But instead of the *VisualWorksPackageImporter* the *CDIFImporter* obviously controls the creation of the FAMIX model based on the CDIF file. We reveal that all the instances of the FAMIX model are created directly from this class. Therefore it is no surprise that *CDIFImporter* is a single-feature class.

- *MSEModel*, *MSEEnumeratedGroup*: As well as in the loading model from smalltalk feature we detect a *tower of instances* of *MSEEnumeratedGroup* and the exceptional entity *MSEModel* which holds the model information.

- *FAMIXClass*, *FAMIXMethod* and others: The FAMIX model built from the CDIF file is the same as in the first feature. It contains again 22 classes and meta-classes as entities of *FAMIXClass*. As the imported CDIF file is the same exported in the previous feature this proves that the export and import from CDIF files works correctly.

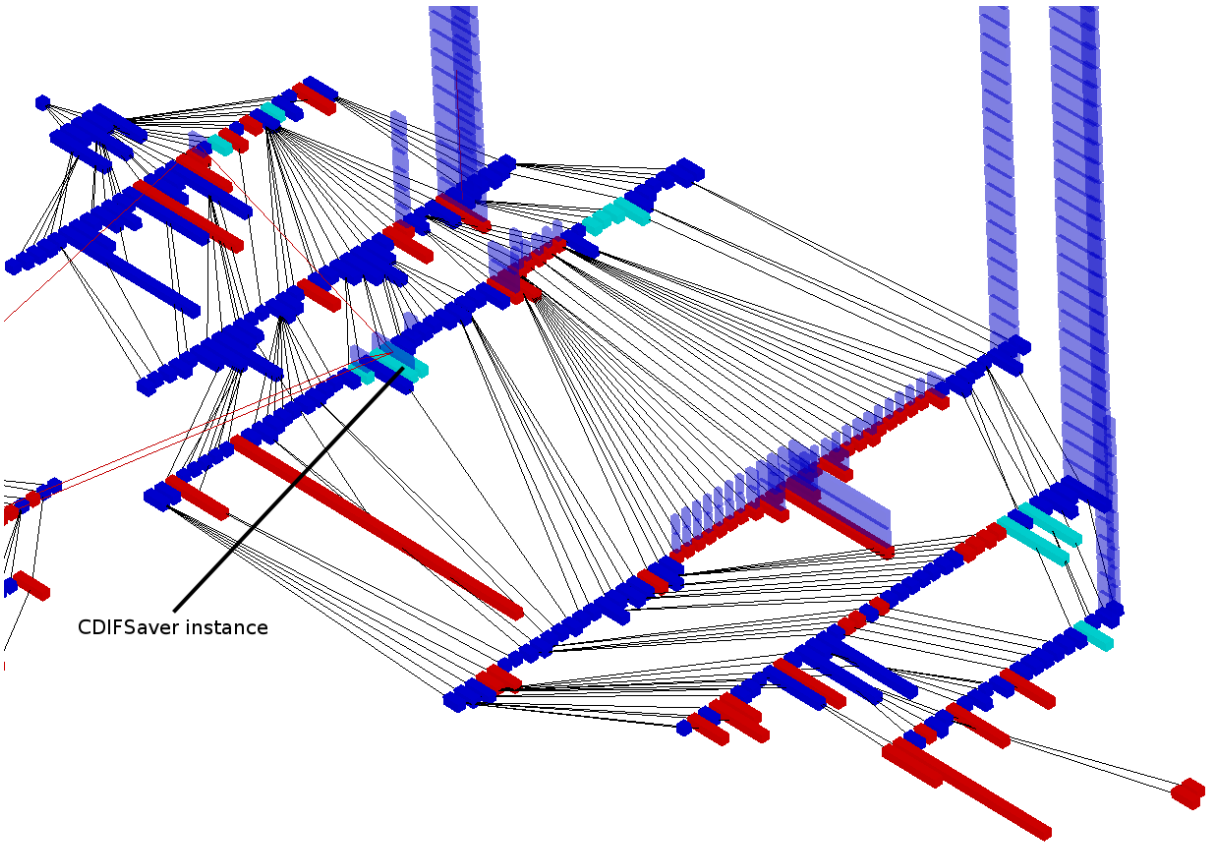

Figure 4.16: The *Model* hierarchy of the export to CDIF feature

4.4.5 Discussion

The interpretation of the features of this case study was significantly more difficult than for the Smallwiki case study. Although we used a divide and conquer strategy and only focused on the most important parts of each feature the visualization of thousands of messages reveal their limit. Nevertheless we were able to determine the most important entities for each feature and were able to analyze their run-time behavior.

This case study also revealed that the detection of parts of the *feature-trace* which are shared by all features can be easily determined by the comparison of the different views. During this case study we detected that the setup routine of the test cases generating the execution trace is the same for all features. The source code of the test case proved this assumption.

The most important conclusion of this case study is that a filter mechanism is necessary to reduce the amount of messages being analyzed. Nevertheless we were confronted with features which use a lot of message invocations during their execution. There we detected the limit of our visualizations. This is mainly caused by the intention of the features as the importing and creation of models is a task which normally leads to a lot of side effects such as the computation of operators, meta-descriptions, etc.. Once we are able to reduce the amount of message invocations to a lower level the view easily revealed a lot of their run-time behavior. Moreover, the feature that exports a model to a CDIF file revealed a further limitation of our approach. We cannot visualize message invocations where the sender or receiver instance is not created within the *feature-trace*. Therefore we would need to analyze the software system before executing the features which we list as future work to improve our approach. We will discuss this limitations in Chapter **??**

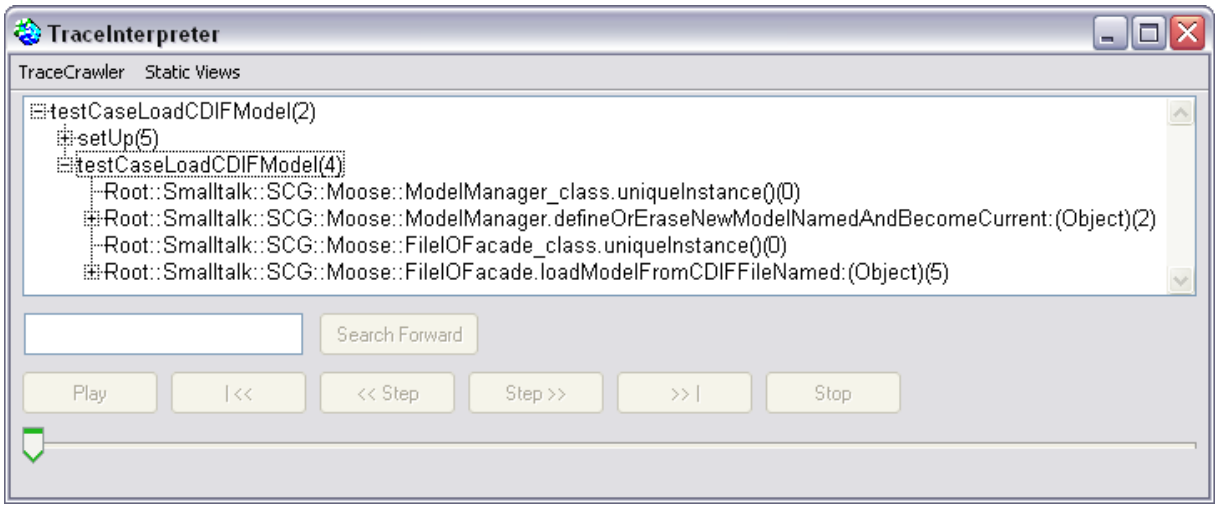

Figure 4.17: The tree view of the import from CDIF feature

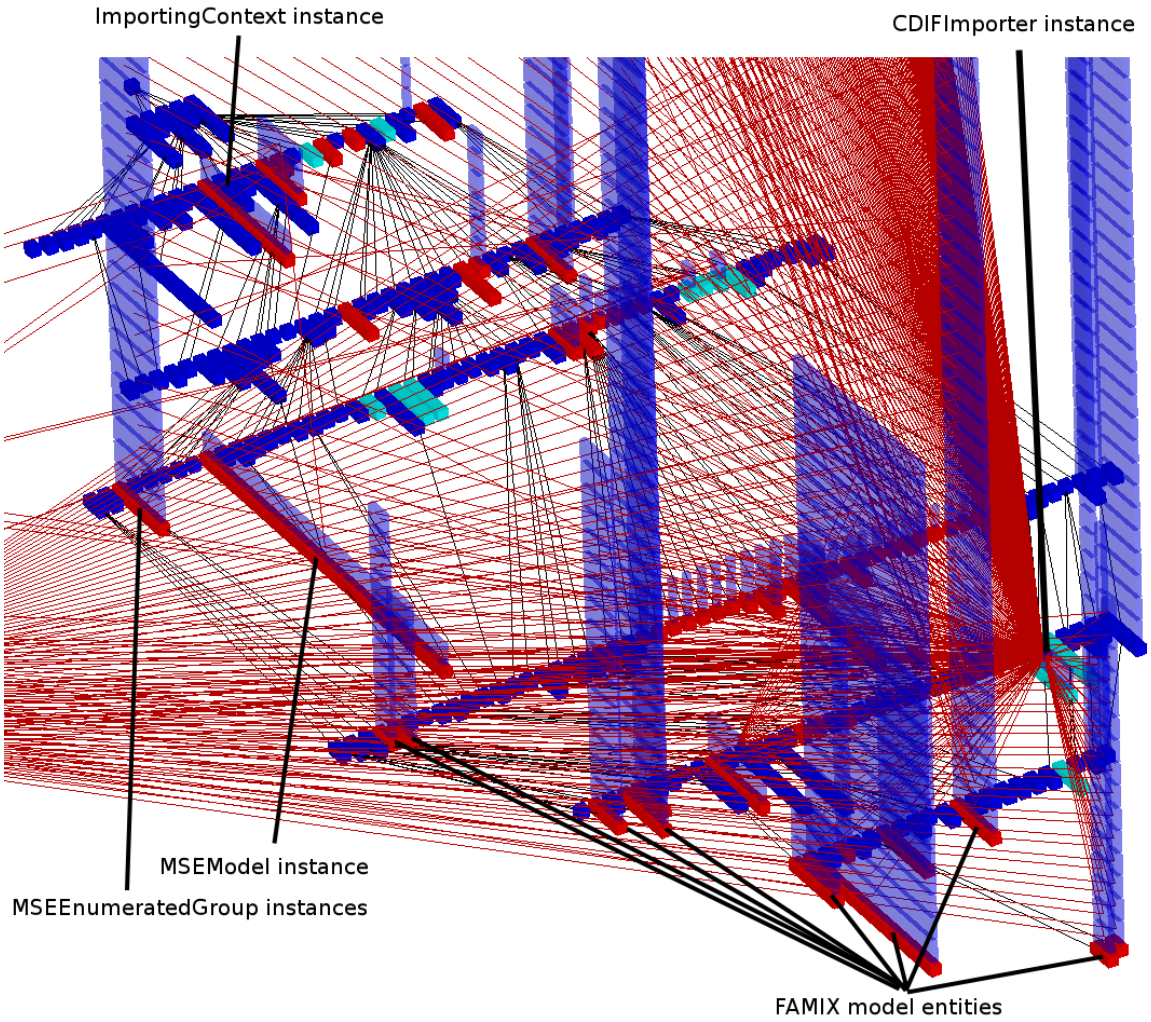

Figure 4.18: The *Model* hierarchy of the import from CDIF feature

Chapter 5

Conclusion

In this chapter we present the results of applying our 3-D visualization of dynamic *feature-traces* by initially answering the question asked in the introduction of this thesis.

- Does our 3-D visualization of feature execution support program comprehension of the dynamic behavior?

  Yes, our views and animation of *feature-traces* provide a visual representation of dynamic behavior of the software system. Our tool *TraceCrawler* allows to zoom in to relevant entities and provides a link to the source code and other visualization techniques such as the *Class Blueprint*. Using this possibilities the software reverse engineer may detect all the entities that participate in a feature. This significantly supports the understanding of dynamic behavior of these entities by navigating through the *feature-trace*.
  However one needs to be aware of the limitations of this approach. It is still difficult to analyze huge feature-traces although the visualization reduces the information significantly. Nevertheless, the analysis especially using the navigation is not easy anymore because of the amount of message invocations in huge traces.

- Which parts of a software system are affected by one or more features?

  Our *static feature interaction view* lets a software engineer detect entities that are affected by one or more features using the colors provided by the FC measurement. With this visualization it is easily possible to determine classes as well as whole inheritance hierarchies that participate within a feature. With this information a software engineer has a significant support to focus on the relevant parts of the software system.

- How do the features interact with each other? Which parts of the system are used by all the features?

  The FC measurement provides an easy way to determine which parts of the system are participating in all features. This measurement is built in all views our tool *TraceCrawler* is providing. In our case studies we showed that we are able to identify feature interaction in terms of classes that are participating in one or more features.

- Can we identify patterns of activity that are shared by features?

  It is difficult to determine parts of the runtime behavior feature are sharing. Comparing the views of the different features may reveal some information of their interaction but does not deliver a secure method to determine their interaction.

- Are there any parts of the system that are stressed? By stressed we mean areas of high activity in the execution of a feature, in other words classes and objects that are sending and receiving a lot of messages.

  Yes. Our *static instance collaboration view* reveals the entities such as class and instances which are communicating more than others. Furthermore, the formation of a *feature hotspot* can be easily determined using our animation of the feature trace. However, the interpretation is difficult as we are not taking the execution time into account. If we were to enhance our animation with execution time information this would lead to a more meaningful conclusion how the parts of the system are really stressed.

5.1 Summary

In this thesis we present a novel visualization technique which combines static analysis of source code with dynamic information extracted by exercising features of a system. This technique will help software reengineers to understand the dynamic runtime behavior of features. We therefore introduced state-of-the-art techniques in the research area of static and dynamic software analysis in Chapter **??**. In Chapter **??** we introduced our views used to visualize the behavior of features. We showed that using our *static feature interaction view* we can easily determine parts of the software system that are active during the execution of different features. Moreover, we showed that we are able to detect the formation of *feature hotspots* which is useful way of understanding the dynamic behavior of individual software entities. As a proof of concept we presented two case studies which show how our approach simplifies the understanding of the dynamic behavior of features. We presented in Chapter **??** the results of these case studies. We describe in the appendix in detail the implementation of our tool in the context of the existing environment of *Moose*, *TraceScraper* and *CodeCrawler*.

5.2 Limitations

Especially during our case studies we detected some limitations of our approach.

- **3-D Navigation:** The 3-D visualization is an important part of our approach. It allows to combine polymetric views with the runtime information collected by feature execution. This results in a combination of static and dynamic analysis which is very helpful to understand the relationship between static and dynamic entities within a software system. Nevertheless the 3-D visualization has also its limitations. To familiarize oneself with the navigation requires practice and some functions as drag and drop, selection of multiple nodes etc. are not implemented in this version of the tool.
- **Scalability:** Although we handle the runtime information in a efficient way and todays computers have a lot of memory, the loading of the information lasts a long time in case of large features. In our *Moose* case study we had to handle almost 100'000 message invocations which leads to an equivalent amount of objects for the visualization. Although our visualizations are very useful to hide this complexity the loading process tends to be slow in such cases.
- **Feature-trace Coverage:** We discovered that we cannot visualize all message invocations of the analyzed *feature-traces*. This is caused by the method wrappers which cannot be installed on the entire system which means that familiar classes such as *String* or *Object* are not included in the *feature-trace*. The reason for this limitation is again the amount information and the execution time needed to run the instrumented system while collecting the *feature-trace*.
- **System Coverage:** Another limitation is caused by our feature-centric approach. We do not start our visualization with an initialized model which is then used to exercise the features on. In our case studies we showed that this leads to an incomplete view of the runtime behavior. How this could be solved is listed in Section **??**.

5.3 Future Work

As this approach is based on a lot of other techniques there is a lot of potential of improving our approach.

- An important extension of our approach would be to enhance the model with state information before executing the features on it. This would lead to higher coverage of message invocation we could visualize. Therefore the system needs to be instrumented from the beginning of the setup process.
- Our approach is language independent but the collection of *feature-traces* is not. To install method wrappers in object-oriented languages such as Java would allow to analyze software system written in other languages than Smalltalk. As soon as someone provides execution traces with the information defined in our approach the visualization will work independent from the language.
- A more sophisticated filtering technique would allow to reduce the amount of messages being visualized. We propose a filter mechanism which is controlled by the software engineer by selecting a group of messages which should not be processed. This would lead to views which can be interpreted much better.
- A current research area is the detection of patterns of activity [?]. The result of analyzing our views to identify such patterns could provide a more high-level analysis of the runtime behavior.

Appendix A

Tools

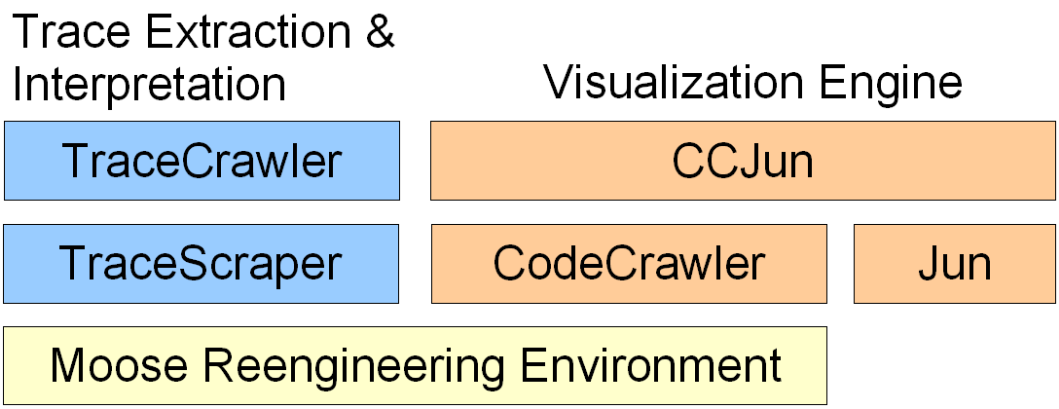

Figure A.1: Illustration of the architecture of our tools

In this chapter we discuss the architecture of our tool *TraceCrawler* and how it is integrated and connected to other tools. In Figure **??** we provide an illustration of the architecture of our tools. The basis of our tool *TraceCrawler* is the FAMIX meta-model and the *Moose* reengineering environment. The feature-trace extraction using code instrumentation is realized by *TraceScraper*. *TraceCrawler* is using the collected information and steering the visualization. The visualization engine is realized by our tool *CodeCrawler* which gets the meta model information from Moose and provides a model for the visualization and 2d visualizations. *CCJun* provides an interface to *CodeCrawler* and is extending its capabilities to the third dimension.

A.1 Feature-trace Extraction and Interpretation

For the feature-trace collection the tool *TraceScraper* executes test cases or scripted scenarios and code instrumentation which records each message invocation and saves it in the meta model of the *Moose* reengineering platform. Our tool *TraceCrawler* then interprets the information in the meta model step-by-step or summarized and controls the visualization.

A.1.1 TraceScraper

TraceScraper provides method wrappers to collect *feature-traces*. A method wrapper captures each message invocation during executing the feature and extracts the necessary information such as sender, receiver and return value. Our tool *TraceScraper* imports the traces and models them as FAMIX entities in Moose. Therefore it extends this meta model to model *feature-traces* as first-class entities.
TraceScraper has various instruments to analyze *feature-traces*. Using a set of features which are compacted to *feature-fingerprints* it facilitate the correlation of features and software entities such as packages, classes and methods.

A.1.2 Moose

The Moose reengineering platform [?] is based on the FAMIX meta model specification [?, ?]. It provides a language-independent representation of object-oriented software systems and instruments to reengineer and reverse-engineer. It supports navigating, querying, metrics and refactorings, etc. of object-oriented source code. The FAMIX meta model comprises the main object-oriented elements such as Class, Method, Attribute and Inheritance as well as Invocation and Access. Our *TraceScraper* tool enhanced the meta-model with further first-level entities to model feature-traces.

A.1.3 TraceCrawler

Our tool *TraceCrawler* interprets the *feature-trace* information stored as FAMIX meta model and controls the visualization. The interpretation is based on the information that our tool TraceScrapers collects by instrumenting the code of the target software system. The objects of the class *ScenarioNode* that are stored as FAMIX entities represent each of them a single message invocation. Using the values of each *ScenarioNode TraceCrawler* creates the visual model by creating nodes (for object instances) and edges (for message invocations) using the *CodeCrawler* tool.
Figure **??** shows the user interface of *TraceCrawler* which can be started directly from the *Moose* reengineering environment. It provides the navigation controls through the feature trace by stepping through it manually or automatically. Apart from using a settings dialog one can control preferences of the animation and there are shortcuts to static 3-D visualizations. Moreover, using a search dialog we provide the possibility to quickly navigate and locate classes or methods within the *feature-trace*.

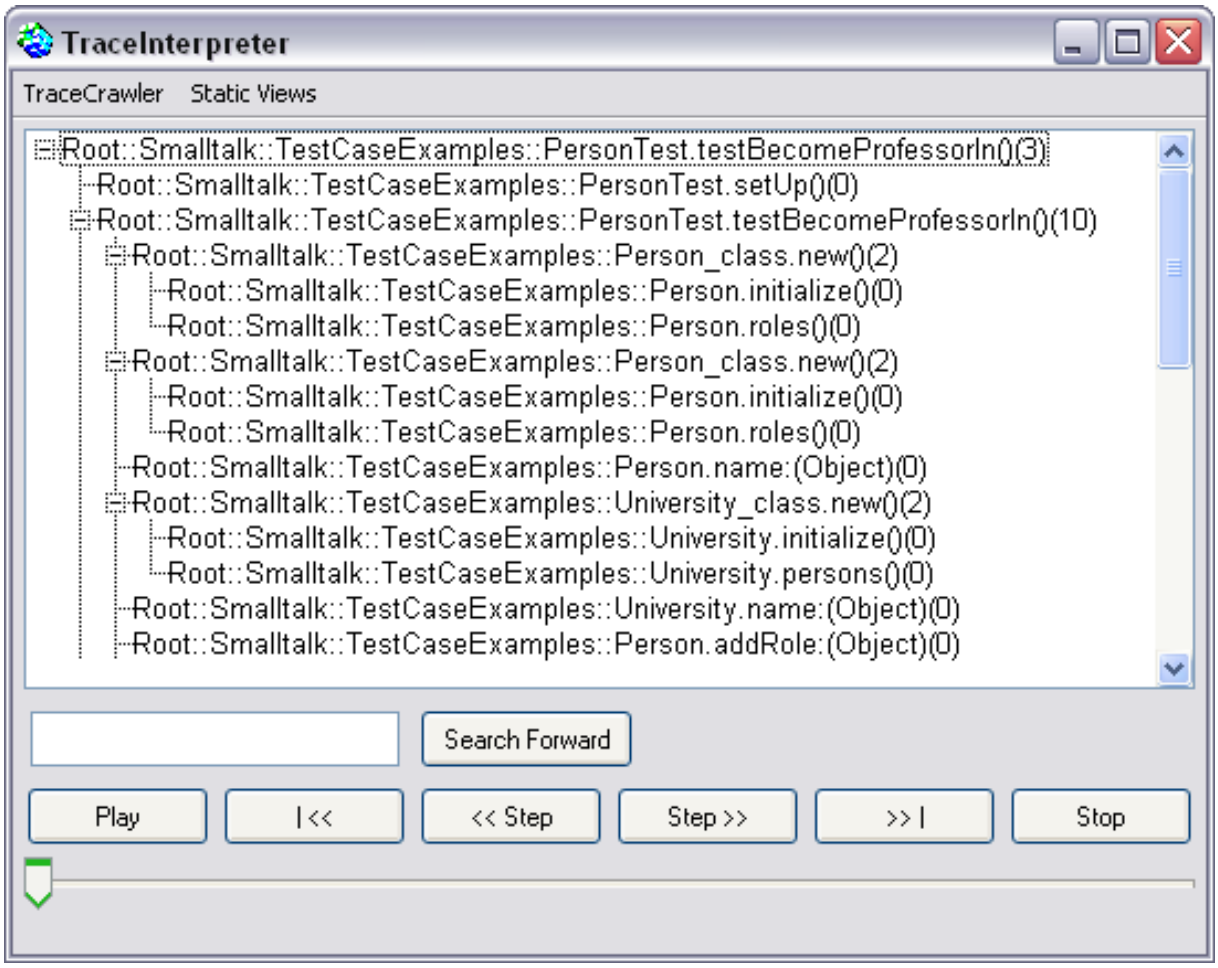

Figure A.2: The user interface provided by TraceCrawler

A.2 Visualization Engine

The visualization engine used by *TraceCrawler* is based on three different tools. Our *CodeCrawler* tool visualizes polymetric views in the 2-D space. It provides the connection to the FAMIX meta model where the information to be visualized is modeled. *CCJun* is our 3-D visualization and animation tool which uses the 3-D graphics library Jun to display the 3-D visualizations and animations.

A.2.1 CodeCrawler

CodeCrawler [?] visualizes polymetric views although it is also a generic information visualization tool. It is built on top of Moose which provides the FAMIX meta-model *CodeCrawler* is using to model its visualizations. It provides a huge variety of different 2-D visualizations for static source code analysis on a coarse-grained or fine-grained abstraction level. It implements different layout algorithms and a lot of other functions that help to abstract different views of a software system.
We use *CodeCrawler* to load the FAMIX meta-model entities into its own model. Using the layout algorithms and other functionality this tool provides the functionality *TraceCrawler* uses to create our novel views.

A.2.2 Jun

Jun[1] is a large 3-D graphics framework with support for OpenGL, VRML and other visualization techniques. Besides other features it provides a hierarchy of classes that allows to create OpenGL objects which are displayed on screen using its own user interface. OpenGL is a good choice because it is available on a lot of platforms incorporated directly in the operating system.

A.2.3 CCJun

CCJun [?] is an extension of *CodeCrawler* which enables it to display 3-D polymetric views using the 3-D graphics library Jun. To support this task it acts as a bridge between those two tools. It provides adapted 3-D objects which can be displayed using the OpenGL implementation of Jun. Therefore *CCJun* adds 3-D figures to *CodeCrawler* to redirect the visualization input to its own engine.

[1] See http://www.srainc.com/Jun/Main_e.htm

Appendix B

Programmers Guide to TraceCrawler

In this chapter we present a short guide to software engineers which would like to use our tool *TraceCrawler*. Therefore we show how to load it, how to generate feature-traces and give some hints to use of it.

B.1 Loading TraceCrawler

Before loading *TraceCrawler* you need to have a VisualWorks virtual machine and an running image which you can download as non-commercial version from the Cincom website[1]. The development and case studies of *TraceCrawler* were done with the version 9.2nc. Afterwards you need to download our 3-D framework Jun which you can obtain from the FTP repository of Cincom[2]. Download the ZIP file and install it according to the instructions for your operating system.

To load *TraceCrawler* you need to connect to the store database on the IAM database server of the university of Berne. The package is named *TraceCrawler* and is dependent on the following packages which will be automatically loaded:

- *AareTraceScraperDevelopement*
- *CodeCrawlerDevelopement*
- *MooseDevelopment*

While loading this packages choose the latest version of each tool. Our case studies was realized with AareTraceScraperDevelopement 4.293, CodeCrawlerDevelopement 4.631, MooseDevelopment 3.0.25 and TraceCrawler 1.71. As *Moose* was refactored during this thesis *TraceCrawler* and *TraceScraper* needs to be adopted to the new version which has not been finished yet. If the loading of the newest version of each tool fails, load the version mentioned above. To ensure that the tools are compatible just run the tests provided in the test package of *TraceCrawler*. This loads a small test model and tests if all the functionality needed to analyze *feature-traces* is working.

[1] http://smalltalk.cincom.com/downloads/index.ssp
[2] ftp://ftp.cincomsmalltalk.com/pub/goodies/Jun/

B.2 Generating Feature-traces and using TraceCrawler

In this section we show how to generate feature-traces for a small example system. Therefore you start your VisualWorks image and open the System Browser. Locate the *TestCaseTraceTest* class and execute the *#testCreateTestCaseTraceExamples* method using the test runner. This installs the wrappers on the test model and then runs the features in the instrumented environment. As a result a FAMIX model with a small test model and four *feature-traces* will be built.

To use the visualization provided by *TraceCrawler*, open Moose and click on the model you wish to analyze. Clicking on the *StarBrowser* icon opens a view which allows to explore all the entities in the current model. Especially you will find there the entity type of *feature-traces* called *TestCaseTrace*. The context-sensitive menu for each *feature-trace* contains a item to open *TraceCrawler*. This action loads the current model using the model of *CodeCrawler* and opens *TraceCrawler*.

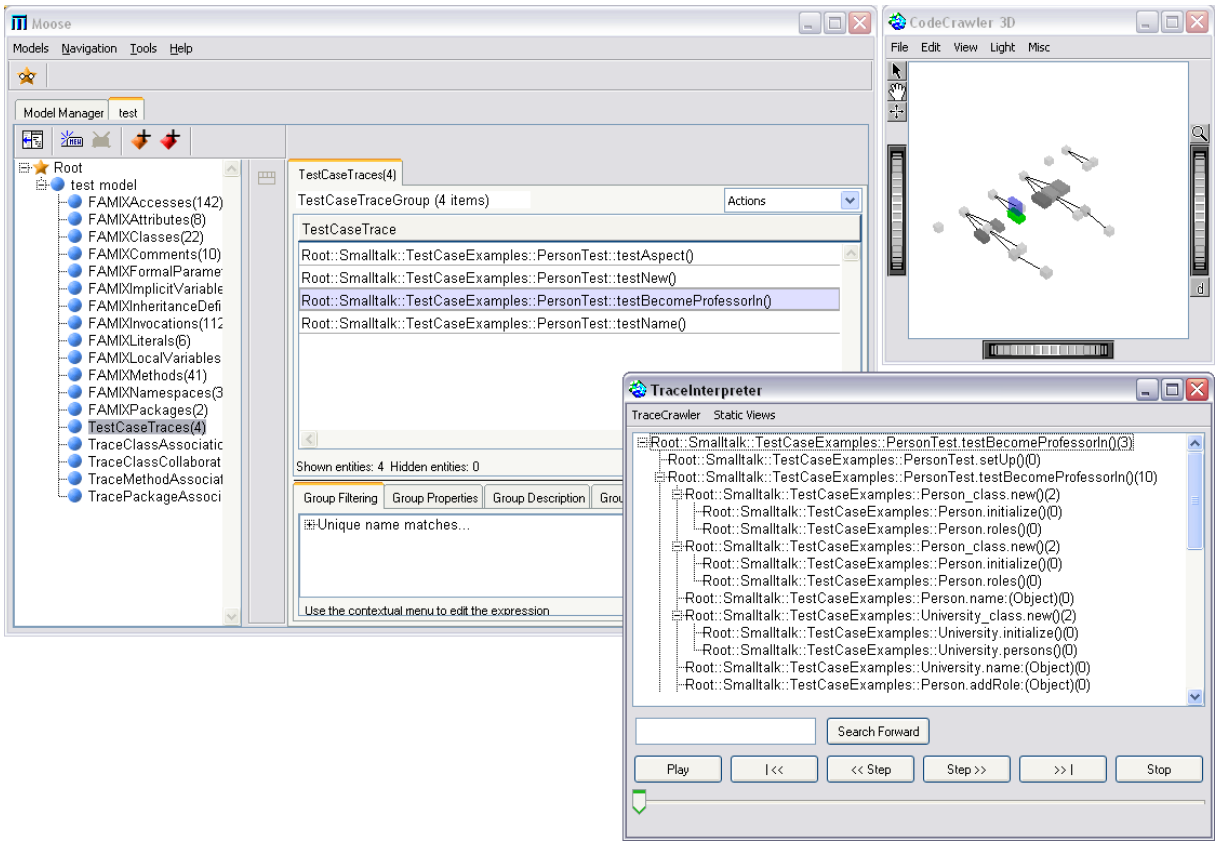

Figure B.1: Screenshot of the *Moose* reengineering environment and *TraceCrawler* in action using a small example system

As shown in Figure **??** *TraceCrawler* initially provides an overview of the *feature-trace* using a tree view. Moreover, the tool provides a menu to open static views such as the *instance collaboration view* and the editor to edit their definition as well as the menu to open the animation of the *feature-trace*. Using the buttons on the bottom the user controls the step-by-step animation. Furthermore using the search dialog the identification of method invocations within the *feature-trace* based on their class or method name is supported. To use the filter technique which lets the software engineer focus on selected parts of the *feature-trace* use the context menu on the node you would like to be the new root node.

Enjoy using *TraceCrawler*!

List of Figures